Johanna Sepúlveda*, Felix Wilgerodt, and Michael Pehl

Towards memory integrity and authenticity of multi-processors system-on-chip using physical unclonable functions

<https://doi.org/10.1515/itit-2018-0030>

Received October 21, 2018; revised January 31, 2019; accepted January 31, 2019

Abstract: A persistent problem for modern Multi-Processors System-on-Chip (MPSoCs) is their vulnerability to code injection attacks. By tampering the memory content, attackers are able to extract secrets from the MPSoC and to modify or deny the MPSoC's operation. This work proposes SEPUFSoC (Secure PUF-based SoC), a novel flexible, secure, and fast architecture able to be integrated into any MPSoC. SEPUFSoC prevents execution of unauthorized code as well as data manipulation by ensuring memory integrity and authentication. SEPUFSoC achieves: i) efficiency, through the integration of a fast and lightweight hash function for Message Authentication Code (MAC) generation and integrity verification of the memory lines at runtime; and ii) lightweight security, through the use of a Physical Unclonable Function (PUF) to securely generate and store the cryptographic keys that are used for the application authentication. We discuss the security and performance of SEPUFSoC for single core and multi-core systems. Results show that the SEPUFSoC is a secure, fast, and low overhead solution for MPSoCs. We discuss the SEPUFSoC security and cost, which strongly depends on the PUF and hash selection. In the future, new technologies may allow the exploration of different PUFs.

Keywords: MPSoC, PUF

1 Introduction

The Internet-of-Things (IoT) provides a huge number of potential new applications enabled through connected smart low cost devices. Maximizing the benefit from the potential capabilities of IoT in domains like industry automation, health care, avionics, and automotive requires computational power as well as flexibility. Multi-Processors System-on-Chip (MPSoCs) arise as an enabler technology

for IoT, providing the required performance and flexibility demanded in such systems. MPSoCs integrate heterogeneous Intellectual Property (IP) hardware cores like processors, hardware accelerators, peripherals, and memory in a single system. They provide the means to implement general purpose or very specific tasks. The capabilities of MPSoCs turn them into a driving force for innovation. As a consequence, they are widely adopted in several critical applications, thus, turning MPSoCs into an interesting target for attackers. Taking over control of a single MPSoC-based node in a critical network can break down its complete security.

Current state-of-the-art MPSoC architectures available in the market, such as Tile-Mx100 from Tiler, MPPA from Kalray, or SCC from Intel [35], integrate a wide amount of IP hardware cores empowered with a deep memory hierarchy, which includes up to L3 caches and DRAM. The IP cores exchange data, wrapped as packets, through a set of buses and a Network-on-Chip (NoC).

The complex architecture of the MPSoC also represents a risk. It opens a wide attack surface. In order to meet the time-to-market pressure, the MPSoC design is based on integrating pre-tested and pre-verified IP hardware cores from third-party vendors. As a consequence, heterogeneous and probably untrustworthy components are integrated in the system, thus becoming a serious threat in the system security. For instance, a malicious communication structure (Bus or NoC) may be able to modify messages. Moreover, code and data of MPSoCs are stored in unprotected external non-volatile memories. Thus, an attacker can modify the content of the memory in order to take over control, extract secrets or to deny the system operation. While Denial-of-Service (DoS) attacks in the context of MPSoCs have been widely studied [15] [36] [31] [6], the memory content manipulation to extract secrets and gain control over the MPSoC remains a challenge.

A common attack to achieve this goal is code injection, where the attacker stores malicious code in the MPSoC's external memory. If the modification is not detected, the malicious code is executed, thus providing the attacker the perfect mean to take control over the system. An attacker may compromise sensitive code and data, to install new firmware or to execute further malicious code.

*Corresponding author: Johanna Sepúlveda, Technical University of Munich, Munich, Germany, e-mail: johanna.sepulveda@tum.de
Felix Wilgerodt, Michael Pehl, Technical University of Munich, Munich, Germany, e-mails: wil.Ger38@net.de, m.pehl@tum.de

To prevent an attacker from succeeding with such attacks, the code stored in the unprotected non-volatile memory needs to be secured. While confidentiality is mostly not required in this scenario, the security of the system can be retained by guaranteeing the integrity and authenticity of the code. These two security services must be continuously guaranteed along operation to ensure a secure MPSoC.

Previous works have proposed different memory protection schemes [28, 42, 26, 32, 14, 23]. Despite the good results, these works have strong requirements and constraints regarding the system characteristics. First, the protection mechanisms are integrated into the micro-architecture of the processors. This prevents the use of general purpose CPU architectures like ARM or X86. Thus, turning these solutions unfeasible for a wide variety of designs. Second, the cryptographic key in such systems is typically stored in non-volatile memory (NVM). Thus, it requires a secure on-chip NVM, which comes with two major challenges: i) the permanent protection of the NVM, that includes the integration of countermeasures and dedicated implementation strategies to avoid passive attacks (e. g., probing, side-channel or fault attacks) and to prevent an attacker from reading the secret while the system is powered down. Such protection can be achieved with expensive and restrictive approaches, such as battery-backed countermeasures [41]; and ii) it incurs into a high cost and logistic overhead to program chip-individual keys for every device.

In order to overcome such disadvantages, in our previous work we have presented SePUFSoC, a Secure PUF-based System on Chip [37]. In this work we recapitulate the results from our previous work and provide new details and insights regarding the requirements of the security relevant building blocks in the system. Moreover, we discuss the design space exploration of SEPUSoC.

SePUFSoC is an architecture based on Physical Unclonable Functions (PUFs). PUFs in this context allow for preventing the need of secured NVM and the consequent drawbacks of high cost, protection overhead, and establishment of per-device keys. PUFs, together with lightweight hash functions are suggested to achieve memory authentication and integrity verification of MPSoCs while keeping cost as well as timing and area overhead low. Since PUF as well as light-weight hash can be integrated into the interface between CPU and communication structure of an MPSoCs, SePUFSoC permits the use of general purpose CPUs as well as of dedicated processor architectures. The security of this flexible and efficient architecture scales with the security of the selected component which allows for a trade-off between security and cost.

In our proof-of-concept implementation of SePUFSoC we use SipHash (proposed in [4]) as a fast and lightweight hash function. A SUM PUF is used along with typical post-processing schemes. It was proposed for the first time in [49]. These PUF components are selected to permit the operation of the proof-of-concept implementation on FPGA. However, our system is not only tested on FPGA, but performance in terms of area, latency, and power is also evaluated for a synthesis of the design in a 65 nm technology of ASICs.

The rest of the paper is organized as follows: Section 2 presents previous work on memory authentication and integrity verification. Section 3 describes the MPSoC and the special requirements in terms of security. Section 4 provides our threat model. Section 5 presents the architecture and functionality of SEPUSoC. Evaluation and results are shown in Section 6, before the paper concludes in Section 7.

2 Related works

In order to provide security of the code and data executed on a System-on-Chip (SoC), encryption of application code/data, stored in an off-chip memory, is widely used [28, 42, 26, 32, 14]. However, this technique does not prevent code injection attacks completely. The lack of authentication and integrity verification allows an attacker to relocate code or perform replay attacks to modifying the program flow or execution state of an application.

In [28, 29] XOM (eXecute Only Memory) is proposed. It uses AES-256 to encrypt memory regions through multiple application-specific keys (*master* and *session keys*). All memory lines are encrypted together with a MAC to additionally provide authentication. Application-specific session keys are stored off-chip, encrypted by the master key. The master key is stored on chip. The main drawbacks of XOM are the vulnerability to replay attacks, allowing the attacker to store authorized code for executing it later and the high performance impact, due the encryption/decryption and authentication of each line by AES. XOM has a high memory access latency, turning it prohibitive for applications with high cache miss rates. Also, XOM requires secured NVM to store the master key on chip. The off-chip key storage forces the integration of memory encryption/decryption at the XOM.

In [43, 42, 44, 45] AEGIS (Architecture for Tamper-Evident and Tamper-Resistant Processing) is proposed. It integrates a One-Time-Pad (OTP) AES-128 encryption and a delay-based ring oscillator PUF for MAC generation

to allow the verification of authenticity and integrity of code and data. AEGIS provides different levels of protection and security modes for different applications by including a secure operating system. This increases flexibility and performance when compared to XOM. The main drawbacks of AEGIS are that i) AEGIS requires extensive changes in the architecture, compiler and operating systems of the SoC; ii) it allows the execution of some part of the malicious code before an integrity violation is detected since verification is done while the code is executed. iii) it has low throughput due to the use of AES for MAC generation and impacts the execution of an application by at least 30 % when compared to the unprotected execution.

CSHIA (Computer Security by Hardware-Intrinsic Authentication) [23] is an SRAM PUF-based architecture. The proposed CPU architecture uses an authentication tree, PUFs, and MACs to perform code and data verification similar to AEGIS. Each line of code and data that is stored in off-chip memory is tagged by a MAC. The MACs can be loaded and stored in parallel to the main memory, by integrating an additional memory for MAC storage and a private data bus for accessing the MACs. This decreases the impact on the performance of the execution of applications. However, it also requires deep SoC and CPU modifications, which turns integration of CSHIA into existing CPU architectures infeasible. Also, similar to AEGIS, some unverified instructions can be executed before an interrupt is generated. Since CSHIA was not implemented, performance and cost cannot be quantified.

All of the proposed memory protection schemes do not support MPSoCs. Although, the same principles for authentication and integrity verification of code and data can be applied, MPSoCs have special requirements. They require parallel execution in multiple and heterogeneous IP cores, high performance, and simultaneous execution of untrusted and sensitive applications on the same platform. SEPUSoC offers a flexible, secure and high performance PUF-based architecture able to be integrated to any general purpose CPU architectures. It offers a solution suitable for MPSoCs and overcomes drawbacks of previous works. Table 1 compares key features of previous work with SEPUSoC.

Memory protection was previously enforced by security wrappers integrated in the interface between the IP hardware cores and the communication structure as in [18, 1, 34, 10, 12, 8, 11, 9, 13]. These firewall-based approaches have shown to be an effective isolation mechanism. They store the security policy (set of security rules) of the system and verify packet-wise the communication rights. When the packet content matches the security pol-

Table 1: Summary of related work and SEPUSoC.

Approach	Authent. Integrity Check before exec.	PUF	Implemented	MPSoC
XOM	No/No	No	FPGA	No
AEGIS	No/No	Yes (RO)	FPGA	No
CSHIA	No/No	Yes (SRAM)	–	No
Firewall	No/Yes	No	FPGA	Yes
SEPUSoC	Yes/Yes	Yes (k-SUM)	FPGA	Yes

icy, the transaction takes place. Otherwise, the transaction is discarded and an alarm is triggered to activate a recovery mechanism. According to the reconfigurability and granularity characteristics of the firewalls, they can be classified as static/dynamic and single/multi-level firewalls. Static and single-level firewalls were proposed in [11, 9, 13]. These firewalls are appropriate for MPSoCs that support fixed and static applications characterized by a single level of criticality. These solutions will fail to protect dynamic and mixed-critically MPSoCs. In order to protect these MPSoCs enhanced firewall management is required. The work of [24, 27] proposed the integration of static and multi-level firewalls. Despite the good results for mixed-critical applications, the dynamic nature of some of these applications is not yet supported. Firewall-based MPSoCs are flexible. They are independent of the processor architecture and the security rules may change during runtime. However, the enforcement of a fine-granular authentication and integrity check for different applications will require the implementation of big firewalls. Thus, turning this option infeasible for some applications.

3 MPSoC description

MPSoCs are tile-based structures which are able to meet a variety of application demands. As shown in Fig. 1, each MPSoC tile is either composed of a single IP core or a cluster of IP cores, which are connected through a communication structure. MPSoCs integrate heterogeneous general purpose CPUs, storage components, peripherals, hardware accelerators, and other IP hardware cores. In order to increase the performance, current SoCs implement memory hierarchies, where several levels of cache (e. g. L1 to L3) and a set of DRAMs are integrated. In case of a cache miss, all cache levels of a processing core are queried until the requested data is found. If no level contains the data, the cache coherency mechanism initiates an access to other distant cores or eventually to off-chip memories.

Figure 1: SEPUFSOC integrated into a MPSoC.

MPSoCs are able to support several applications, usually being executed simultaneously. For performance enhancement two approaches are used in MPSoCs: i) memory hierarchies, which use the spatial and timing locality principles; and ii) resource sharing, which allows to split applications into small pieces of code (tasks), which are mapped to different MPSoC resources. Code and data of the applications are normally stored into off-chip memories. Thus, avoiding code injection attacks is a serious security concern for MPSoCs.

4 Threat scenario and scope of SePUFSOC

MPSoCs which are connected to the internet are vulnerable to attacks which exploit security issues in the communication protocol or bugs in the software. Although such attacks are not in the direct focus of SEPUFSOC, these scenarios can lead to corrupted cores which are used to inject malicious code, a scenario which is discussed below. We assume the components integrated inside the MPSoC are trusted. That means that the communication among the components is secure. The works of [40] [39] [38] discuss the secure operation of MPSoCs where untrusted communication structure is considered. However, this is out of the scope of this paper. Also, invasive and semi-invasive physical attacks against on-chip components such as processor cores or caches during operation of the system are out of the scope of this work. Off-chip memories are a critical component of the embedded system. Applications are stored in such memories and the SoC reads the data and code from this memory or write back data. One of the most severe attacks on SoCs and MPSoCs is the injection of code into off-chip memory. This memory is specially vulnera-

ble. Due to cost reasons, such memory is typically not protected. Thus, an adversary might exploit this fact to write malicious code or data into the off-chip memory, reprogram the memory, or remove parts of it. The code is later executed or used by the MPSoC and can lead to an infection of the complete system. Similar argumentation holds for other off-chip components like peripherals which are connected to the system and buses, like the memory bus which connects the MPSoC to the external memory. Without protection mechanisms, the MPSoC cannot distinguish malicious from innocuous messages and an attacker might use unprotected interfaces to infect the system.

To prevent an attacker from injecting malicious code, authenticity and integrity of the code needs to be guaranteed. The fulfilling of this requirements incurs an overhead of testing the authenticity and integrity of code during operation and additional storage of authentication data. Note that proving authenticity and integrity needs to take place before the code is executed. We assume that confidentiality of the code is not an issue (otherwise, additional means to encrypt the code in the unprotected memory are required). Also, denial-of-service attacks are out of the scope of this work.

In such a setting, where code authenticity and integrity is required, new application code and data from peripherals can be downloaded to the system by the manager IP core of the MPSoC. This instance, referred in this text as hypervisor, validates the integrity and authenticity of software or data with classical means. However, classical signature schemes, such as asymmetric cryptographic algorithms, do not provide sufficient performance for being used during execution of the code. Also, since not all code might be loaded into the system at once, the authenticity and integrity of pieces of codes might be proven before its execution on the MPSoC. In this sense, when adding new code to the external memory (e. g., new application is

stored in the off-chip memory) or when storing data during operation, the hypervisor and the IP hardware core that is responsible for executing the task, should be able to sign the code and data in a line-basis and write back this information to the off-chip memory. The signing should be performed using either a dedicated key or a key which is common for all or certain components in the MPSoC. The authenticity of code and data is validated by a component which has access to the external memory, upon loading.

SEPUFSoc provides an efficient and flexible mechanism to enable such integrity and authenticity checks of code and data during operation. Memory modification attacks such as code injection and related attacks are prevented by this scheme. Especially, SEPUFSoc can be used to prevent spoofing, reallocation and, to a certain extent, replay attacks to off-chip components.

4.1 Key-storage with PUFs

SEPUFSoc relies on Physical Unclonable Functions (PUF) based key storage. Thus, the basics of this concepts are recapitulated in this section.

PUFs measure uncontrollable process variations that appear during the chip manufacturing. These variations are unique for each chip and cannot be extracted by an adversary. Thus, a good PUF implemented on different chips generates a different response for each chip that is unpredictable from outside. These responses are used as a key, which is stored in RAM so that the secret (i. e., key as well as PUF responses) cannot be accessed when the chip is powered off. Thus, active countermeasures need to be implemented only for attacks during operation of the chip.

The advantage of the PUF responses' volatile nature comes with a downside: noisy responses. The PUF response is the result of a measurement process, thus, the PUF might provide a different response for different readouts depending on environmental, aging and operational conditions. Currently, an error rate of up to 15 % or even 25 % per PUF response bit is considered. However, the error probability reported in previous works for a key which is derived from a PUF is typically below 10^{-6} or 10^{-9} [30, 20]. Thus, the integration of error correction capabilities is required.

Error correction mechanisms require that a code word (i. e., a bit string with a certain structure) needs to be provided at the input of the error correction algorithm. Since the PUF response is a random pattern without structure, it needs to be mapped by a so called *helper data* algorithm to a codeword of the used code.

Thus, to store a new secret, the following steps are required:

1. A secret S must be derived. Where S can be a random number generated by a true random number generator or a part of the PUF responses.
2. A PUF response is generated and a helper data algorithm is applied to map S and PUF responses X which are not used as the secret to helper data W . In case of a *Fuzzy Commitment* scheme [25] this is achieved by encoding S to a codeword C and XORing X and S to receive $W = X \oplus C$.

To reconstruct the secret,

1. The PUF responses $\hat{X}$, which were used to create the helper data, are generated. Note that due to noise and other environmental factors, the actually received responses in this step can slightly differ from X .
2. The $\hat{X}$ and the helper data W are mapped to a noisy codeword $\hat{C}$. When a Fuzzy Commitment scheme is used, this process is performed by XORing the helper data and the PUF responses to $\hat{C} = W \oplus \hat{X}$.
3. Finally, the noisy codeword is decoded to $\hat{S}$. For a sufficiently good code $\hat{S} = S$, thus, the secret is restored.

Note that for a good PUF, the helper data does not provide any information about the secret [33] [7]. Thus, it can be stored as public information in an off-chip memory. Moreover, the same helper data applied to any other PUF will lead to a different secret. This binds the secret, which can be reproduced with a given set of helper data, to a certain chip. Furthermore, the process of extracting a secret from the PUF allows to generate different keys on different devices, and does not require expensive on- or off-chip infrastructure.

In case that the PUF response has full entropy, the secret can be used directly as a key. Otherwise, the helper data may leak information and thus should be further processed. If the entropy per bit in S should be below 1, a hashing algorithm is used to compress a longer secret vector S into a key K such that the required entropy per bit is achieved. This process leads to a large-scale design space exploration, including different error correction and helper data schemes, together with different PUF types can be used. According to the way that the secret is extracted, PUFs can be classified into two categories. First, the so-called Single-Challenge PUFs, that includes PUFs with no challenge-response behavior. SRAM PUFs belong to this category, where a certain PUF cell (e. g., SRAM cell) is expected to provide always the same response. Second, the Multi-Challenge PUFs, where PUFs with challenge-

response behavior are used. The response of the PUF is strongly dependent on the challenge.

In the first case, multiple keys can be generated from the PUF, e. g., by using a sufficiently large array of SRAM cells and selecting different parts via an address space. In the second case, besides the helper data, also the challenge needs to be stored as public information. Different challenges and helper data applied to the same PUF lead to different secrets. The selection of the PUF and associated components is further discussed in Section 5.3.2

5 SEPUFSoc

The main goal of SEPUFSoc is to ensure memory authenticity and integrity. To provide a high level of flexibility and in order to be compatible with different architecture structures, an intermediate layer is added between the bus and the IP hardware cores. The bus is used for the communication among the IP hardware cores and among The IP cores and the off-chip memory. We call this intermediate layer *authentication controller*.

The overall MPSoc structure is shown in Fig. 1: The N CPU cores of the MPSoc, which might be general purpose CPUs, are interfaced to the bus through the authentication controller (*Auth. Cntrl.*). Each CPU is equipped with its own instruction and data cache, while applications and data are stored in off-chip memory. A central PUF instance provides the key to all authentication controllers via a separate, trusted connection. We assume that this key-transmission within the chip cannot be attacked as well as that the PUF cannot be read out during operation of the chip. Such an assumption is valid, if active countermeasures, like probing attempt detectors [46] and sensors are used to prevent attacks during runtime.

We require two main properties for the authentication controller in this work:

1. **Code and data binding:** Application code and data in the off-chip memory should be bound to a certain device. This property ensures that code from one malicious MPSoc cannot be transferred to another MPSoc. Also, it prevents code injection into memory.
2. **Low latency:** The timing overhead of the memory authentication and integration check should be as small as possible. This is required to reduce the performance impact of the additional security mechanism.

In the presented concept, a keyed lightweight hash-function (the SipHash) provides a sufficient security level while keeping the latency low. Binding of data and key to

a certain device is realized through a key derived from a PUF and thus, processing the data together with this chip-individual key to a message authentication code (MAC). Note that SEPUFSoc allows the selection of any PUF and hash function in order to meet the different system requirements. For instance, low latency might be traded for a higher level of security by generating a longer MAC and using a higher number of SipHash rounds or by using a stronger cryptographic algorithm to produce the MAC.

5.1 Operation phases

Three operation phases are distinguished in SEPUFSoc: i) initialization of the system; ii) installation of new applications to the off-chip memory; and iii) processing of code and data provided through the off-chip memory.

5.1.1 Initialization of the system

After manufacturing of the system, a tiny layer of code and data needs to be installed in a trusted environment. This code must allow later secured installation of application code and the secure transmission of data. Since we consider the installation of application code as not timing but security critical, we suggest to provide in this phase classical asymmetric cryptographic algorithms, like RSA, or even post-quantum secure algorithms. Also, a certificate to verify the authenticity of at least one data source needs to be provided in this step. For more details regarding the secure initialization of the system, the reader is referred to [2]. The code can be stored in off-chip memory and can be already protected by the same mechanism of SEPUFSoc to install new applications. Depending on the use case, also a pre-shared key might be stored via the PUF in this step by classical means of key storage with PUFs. After installation of this initial code and data in a trusted environment, the access to the upload of unauthenticated code can be permanently removed, e. g., by burning a fuse.

5.1.2 Installation of new applications

The process of installation of new applications is shown in Fig. 1. It is composed of five main steps. Code or data may be stored to off-chip memory (step 1). Two cases can be distinguished: i) Either the on-chip CPU wants to write-back data; or ii) the CPU receives data via a public channel and needs to verify (and probably decrypt) the data before storing it into the memory. In the latter case, classical crypto-

Figure 2: Controller architecture.

graphic algorithms are used to implement the security services. Thus, we assume that the CPU always stores trustworthy code or data.

To generate individual keys per application, a single, central PUF unit is suggested in SEPUSoC. This PUF can be realized by a single-challenge (SC) PUF, e. g. an SRAM PUF or RO PUF, or a multi-challenge (MC) PUF, e. g. an Arbiter, Bistable Ring, or SUM PUF. If a sufficiently strong MC PUF is implemented, several thousand to millions of secret response bits can be extracted from it. Such a huge amount of secret bits allows to generate keys for a large number of applications. If a small number of PUF response bits is sufficient, an SC PUF can be used. In particular, the start up behaviour of regular SRAM, which is typically available on every chip, can be used to extract a secret. If the number of keys extracted in this way does not fit the number of application keys, we suggested to modify SEPUSoC: A single key stored by the PUF can be used as a master key. The application keys can then be encrypted by the master key to be stored as public data which are decrypted on demand. This can be recommended together with an SRAM PUF for IC implementations of SEPUSoC.

The data storing process requires that data is first processed by the authentication controller, as shown in Fig. 2. The *Control Unit* requests a key from the PUF (step 2). In this step, a single key, per-core or per-application keys can be used. The control unit determines the correct key and requests it from the PUF. SEPUSoC provides, thus, the flexibility to have either a common key for all cores and applications, or per-core and per-application keys, where the number of used keys determines the requirements for the PUF, as discussed in Section 5.3.

If the control unit cannot find a corresponding key (e. g. if a new application is installed in a per-application

key setting), the control unit requests a new key from the PUF. That is, a random secret as well as a random challenge (for Multi-Challenge PUF) or an yet unused PUF address space (for Single-Challenge PUF) is selected and the corresponding helper data are generated according to Section 4.1 (step 3). Helper data and challenge are stored to the off-chip memory (step 5), where each memory line can be protected by a MAC, to detect data manipulation. This MAC can be generated by the control unit using the key, the data, and the keyed hash function, SipHash in our case (step 4).

If the control unit finds a corresponding key, it loads the helper data and (if applicable) challenge or PUF address space from the off-chip memory. The data are provided to the PUF which responds the corresponding key. Note that in case that this key was used to generate a MAC for the helper data, only now the authenticity of the helper data can be validated since the key is available to the control unit only after receiving it from the PUF, as discussed in Section 5.1.3.

After receiving a valid key, the control unit writes the actual data, which the CPU wants to store, into the off-chip memory. Like described for the helper data, a MAC for each memory line is computed from the data and the MAC using the key derived from the PUF. Also the address of the memory line is comprised in the computation of the MAC to prevent reallocation attacks.

Note that under the assumption of a good PUF, the attempt of an attacker to transfer the stored data to another device or to manipulate the data can be detected.

5.1.3 Processing code and data

In this operation mode, a CPU core requests data from the off-chip memory. As shown in Fig. 1, this request is provided to the authentication controller (step 6). Similar to the installation of new applications, the authentication controller first needs to request the key from the PUF (step 3). Since the code or data have been stored at some point in time before, the authentication controller can always load the corresponding helper data and (if applicable) challenge or PUF address space from the off-chip memory. These data are provided to the PUF and a key is generated according to Section 4.1.

If required, first the helper data can be validated. Afterwards, the actual requested code or data is loaded from the off-chip memory. The code is loaded memory line by memory line together with the corresponding MAC. Every memory line is provided to the keyed hash-function in order to compute the MAC. The MAC computed from the

loaded data is then compared to the MAC stored with the data by the *Compare* unit in Fig. 2. If the two MACs do not match, an alarm is triggered via an interrupt. This alarm informs the CPU regarding the attempt to load inconsistent or manipulated code. Further loading of data is blocked in this case.

5.2 Anti-replay mechanisms

SEPUFSoc provides a means to prevent many attacks that try to manipulate the content of the off-chip non-volatile memory. But until now, replay attacks have not been considered explicitly. Different forms of replay attacks can be distinguished. In the most general case, an attacker takes the content stored at a specific memory address and tries to place this code to a different location in memory. Since SEPUFSoc processes not only the memory line but also the memory address to a MAC which is validated, this type of replay is not feasible in the system.

However, an adversary can read out one or multiple memory lines from off-chip NVM. Later, when different content is stored at this location, the adversary can replace whatever is written at this position with the old content and MAC. In the current version of the system, this would not be recognized. However, depending on the use-case SEPUFSoc can be extended for protection against such an attack.

5.2.1 Protection of sequential operations

SEPUFSoc can be easily extended to prevent an attacker from replacing single lines of code in a sequence of lines. So far, the MAC is computed only per memory line. However, the computation of the MAC can easily be extended to incorporate the hash-value from the previously loaded memory line. This can be done for all lines of code which are processed one after the other given a flag to indicate when such a sequence starts.

However, if data are not processed one line after the other, e. g., if jumps are required or data are read in an arbitrary order, the application controller would require to recompute or load the hash value from the previous lines. However, re-computation in this context is hardly feasible in general, since also the MAC of the previous lines might depend on other code lines so that the effort is too high. Thus, loading the code without authentication would open the door for an attacker. Buffering the hash values from an application and computing missing hashes on demand might be a solution but it incurs a significant

performance overhead in terms of huge on-chip buffers and additional loading times.

5.2.2 Protection by session tokens

A session token is a unique identifier to identify the session. One or multiple valid session tokens can be incorporated in the MAC without significant performance overhead. Session tokens are periodically renewed, e. g., every session replaced and code lines which are authenticated under the utilization of the token need to be signed again. They are loaded by the authentication controller, the current MAC is validated and a new MAC is computed for the new line, same memory address, and new session token before the memory line as well as the MAC are stored again.

The required replacement of the MACs can be done during idle times of the system and the system can be kept awake until the process is finished. The performance overhead in this case is mainly the power consumption for the re-computation of the MAC.

However, the session token needs to be stored in non-volatile memory. If this is done in the off-chip memory and signed using a PUF-based key which is generated using helper data that are also stored in off-chip memory, an attacker is not able to replace single parts of the memory.

However, an attacker can still take an image of the off-chip memory. When the memory content is later restored in the memory by the attacker, the system has no means to identify that old content is provided.

In current state-of-the-art technologies, there are only two solutions: 1) the session token can be stored in secured NVM, which is the best choice in terms of performance but it is not available in low cost devices and might render the use of a PUF in the system useless; and 2) the session token is provided by a trusted third party and transmitted to the system on demand, which has the drawback of additional infrastructure and communication overhead that, however, might be acceptable in certain scenarios. CMOS-compatible emerging technologies might provide a third and better solution in future. PUF cells with an inherent non-volatile behavior have been suggested in this area, e. g., [3]. It was also explored that the cells can be brought into a state where the stored secret cannot be read out by an adversary. But the secret can be recovered on demand. If in future such cells provide in addition a mechanism to generate or store on demand new random values in the same cell, this value is an ideal candidate for a session token: An attacker cannot guess or read out the value due to the PUF-property and, thus, cannot program or en-

force the required value for the token. But the value can be used directly as a session token which can be replaced on demand. However, the development of such mechanisms in new technologies requires additional research in that domain.

5.3 Selection of components

The flexibility of SEPUSoC is originated from the intermediate layer implementation (wrapper-like). The authentication controller is integrated between the bus and the cores in an MPSoC. This controller is suggested to be designed for low cost and low latency but can be adapted to the needs of the MPSoC. The selection of two components mainly determine the performance (throughput, latency), area, power consumption, and security level of the authentication controller: The keyed hash function and the PUF.

5.3.1 Selection of hash function

To enable authentication and integrity checks in the authentication controller, a keyed hash function is needed. SEPUSoC only requires a key for a symmetric cryptographic algorithm. The key length is suggested to be chosen to 128 bit or longer so that it is considered cryptographically secure.

In SEPUSoC, the off-chip memory is authenticated line by line. Compared to signing only a complete application or data structure, this line-wise authentication has the benefit that a corrupted memory line can be identified immediately before it is loaded and not only after loading the complete program. Signing a complete application only would lead to a large latency and require a huge on-chip buffer if the authentication is done before the code is executed or would allow for the identification of malicious code only after its processing. Thus, the hash function in SEPUSoC must support the generation of MACs for short messages. The length of the MAC can be adapted to the required security level. We decided to use a 64-bit MAC in the proof of concept implementation below.

One possible hash function which was developed to generate MACs for short messages with low latency, and which is used in our proof of concept design below, is the SipHash [4] or, more accurate, SipHash- c - d shown in Figure 3. It takes an arbitrary length input message and generates in our case a 64-bit MAC using an 128-bit key, where the message is padded to a length which can be divided into 64-bit chunks.

Figure 3: SipHash- c - d architecture for $c = 2$ and $d = 4$.

To generate a MAC, the key k used for SipHash is first split into two 64-bit chunks k_0 and k_1 , so that $k = k_0 || k_1$. The state of the hash function $v = v_0 || v_1 || v_2 || v_3$ is initialized using the 64-bit words

$$\begin{aligned} w_0 &= 0x73\ 6F\ 6D\ 65\ 70\ 73\ 65\ 75, \\ w_1 &= 0x64\ 6F\ 72\ 61\ 6E\ 64\ 6F\ 6D, \\ w_2 &= 0x6C\ 79\ 67\ 65\ 6E\ 65\ 72\ 61, \text{ and} \\ w_3 &= 0x74\ 65\ 64\ 62\ 79\ 74\ 65\ 73 \end{aligned}$$

as $v_0 = k_0 \oplus w_0$, $v_1 = k_1 \oplus w_1$, $v_2 = k_0 \oplus w_2$, $v_3 = k_1 \oplus w_3$. From the message which is processed, always a 64-bit chunk m_i are taken. Chunk m_i is processed by XORing it to v_3 , applying the SipRound c times and XORing afterwards m_i to v_0 . After processing the last 64-bit chunk, $0xFF$ is XORed to v_2 and the SipRound is applied to v another d times before bit wise XORing together $v_0 \oplus v_1 \oplus v_2 \oplus v_3$ to get the 64 bit MAC. Each of the SipRounds is an add-rotate-xor (ARX) operation, consisting of 5 additions, 6 rotations and 3 XORs.

The strength of the SipHash scales with c and d . However, the authors claim that a SipHash with $c = 2$ and $d = 4$ provides the best possible MAC security which can be achieved by any function with same key and output size. Thus, we can assume that it is not feasible for an attacker to retrieve key material from knowing the messages and MACs in our design.

5.3.2 Selection of PUF and error correction

The quality of a PUF is typically evaluated in terms of its unpredictability and reliability, where both should be as high as possible in case of the PUF integrated in SEPUSoC. The unpredictability determines the strength of a PUF to resist an adversary who tries to extract the key by exploiting weaknesses in the PUF like bias, correlations, or higher order dependencies of PUF response bits between different positions on a device [48], between devices, or between challenge-response pairs of the same PUF. While the first two weaknesses allow the utilization

of smart guessing strategies, the last might not only allow the smart guessing but also the use of machine learning techniques on the PUF.

The reliability of a PUF determines which amount of the entropy from the manufacturing variations can be used and defines the error probability at the input of the error correction algorithm used to derive a key. It defines the required error correction capabilities and, thus, the complexity of the algorithm to derive a key from the PUF.

There is a wide variety of PUFs. Since the PUF should be integrated into the MPSoC while achieving low cost, only silicon-based PUFs are currently a reasonable choice. However, in future also PUFs built in emerging technologies might be used. As previously discussed, PUFs can be either Single-Challenge (SC) or a Multi-Challenge (MC) PUFs.

For SEPUSoC, the selection of the PUF type depends on the actual use-case. For FPGA implementations, e. g., RO based PUFs like RO-PUFs, SUM PUFs and k -XOR SUM PUFs are well suited. However, they have a higher power consumption than other PUF types, so that for ASIC implementations, PUFs like Arbiter PUFs or SRAM PUFs, which can hardly be implemented on FPGA in good quality, might be preferable. SRAM PUFs, for instance, which exploit the startup behavior of SRAM cells, have the additional benefit, that they can be easily implemented in systems with minor or no modification of the system. Instead, the start up behaviour of regular SRAM, can be used to extract a secret, e. g. [47].

For SEPUSoC, more than one key might be needed, e. g. a separate key for every application can be useful in certain scenarios. Due to the redundancy required in the error correction to compensate for noise in the PUF response, for a single 128-bit key about 1000 to 3000 response bits from the PUF are required. For an SRAM PUF scenario, this is equivalent to about 0.4 kB of SRAM per key. For a SUM PUF implementation, a single SUM PUF with 128 ring oscillators might be sufficient. To derive n keys, the amount of SRAM cells needs to be linearly increased. For SUM PUF based implementations, a k -XOR SUM PUF is suggested if the number of keys is larger than one. Such an implementation might scale better in terms of provided entropy for constant, small k , since it has been shown that the number of challenge-response pairs required to machine learn an XOR PUF scales polynomial with k [16] [17]. But the number of XORs and thus the number of keys which can be extracted from such a PUF is limited by the noise which also increases with k .

Another parameter to be considered in the selection of the PUF is its speed. For example, measuring the ring oscillators in an RO based implementation takes several

hundreds of clock cycles;¹ The result of an SRAM cell is available directly after power up and only needs to be read out. Thus, it might be beneficial to use PUFs like SRAM PUFs if speed is a concern. However, the RO frequencies can also be stored in RAM during boot up of the system to speed up the generation of PUF responses or the required keys can be pre-computed and stored in dedicated on-chip RAM. Such a scheme is used in our proof-of-concept implementation.

Depending on the noise characteristics of the PUF, an appropriate error correction code is needed. Strong error correction is required to handle large bit error rates in the PUF response and obtain very low error rates in the key. This is typically achieved with concatenated codes which are sufficiently powerful while keeping the decoder complexity low. Examples are the concatenation of repetition and BCH codes [30] or the concatenation of Differential Sequence Coding (DSC) and Viterby codes [21], the latter also used in our proof of concept implementation. Such codes require at the input a code word. Since the output of a PUF is, however, a random number and has not the structure required for a certain code, Helper Data Algorithms (HDAs) are used like described in Section 4.1 to map the PUF response to a code word.

Two security critical points have to be considered when using HDAs and error correction codes with PUFs: It needs to be ensured that 1) the helper data which are stored in public non-volatile memory do not leak any information and 2) manipulation of the helper data does not reveal information of the secret.

The first requirement can be ensured mainly by a careful choice of HDA and error correcting code. Especially, storing reliability information for PUF response bits in the helper data can be critical information. E. g. in [5] a machine learning attack on a SUM PUF implementation is described which explores the fact that the helper data – generated by index based syndrome coding (IBS) in that particular case – leak information about the PUF response and allow, together with the used challenges, to dramatically reduce the key entropy without knowing any PUF response. Note, that, since we use DSC in our proof-of-concept implementation, which selects only reliable PUF bits, a similar attack might be feasible as long as a SUM PUF is used. However the attack can easily be countered by implementing an XOR-SUM PUF which can be done within the same area by XORing subsequent response bits (leading to a doubled run time) or with the same time overhead

¹ Longer measurement times of ROs correspond to higher robustness against noise and a trade-off between measurement time and reliability can be made.

by XORing the responses of two different SUM PUFs (leading to a doubled area).

Another issue regarding helper data leakage are bias or correlations in the PUF bits. An attacker who knows about such weaknesses of the PUF can gain knowledge about the secret by observing the helper data. Again, appropriate coding strategies can be used to counter the leakage in such a case [22]. However, the most common approach is to generate a longer secret from the PUF, to accept that an attacker might gain limited knowledge about the secret, and to hash the secret to a shorter key with full entropy so that an attacker does not learn about the key.

The requirement that manipulation of helper data does not reveal information regarding the secret is an issue for very specific HDAs, only. E. g., if pointers are stored to select the address of reliable bits (e. g. in an SRAM PUF), exchanging the pointers will lead to the same response if and only if the two response bits are the same. Thus, an attacker might try to redirect the pointers in order to compare bits of the secret. The information used by the attacker in this case is if the same key is computed although different helper data are used. A similar mechanism has been suggested to reveal the secret when DSC is applied.

To prevent helper data manipulation it can be ensured that every modification of the helper data leads to a modification of the key. This can be achieved by hashing the helper data and processing (e. g. XORing) the hash together with the corrected secret from the PUF to the actual key. We use a SPONGENT hash function in our implementation below to ensure that helper data manipulation is not feasible.

However, in case of SEPUFSoC an alternative approach can be taken: The helper data are stored in off-chip NVM. I. e., it can be protected by the authentication controller with a MAC as described above. This does not mean that helper data manipulation is not possible, since the integrity of the helper data can only be checked after retrieving the key. But the validity of the helper data can be checked after computing the key. Thus, if the correct key is computed with the wrong helper data the authentication controller can still decide to drop it so that an attacker cannot distinguish between a wrongly computed key and a correct key derived with the wrong helper data. Therefore, SEPUFSoC provides a means to protect itself against helper data manipulation attacks.

6 Evaluation of SEPUFSoC

The SoC/MPSoC based SEPUFSoCs shown in Fig. 1 are implemented on the Nexys4 DDR FPGA with Vivado 2015.3

Design suite from Xilinx. It is composed of 5 IP cores: i) single or multiple 5-pipeline deep Microblaze 9.5 running single or multiple testbenches; ii) private instruction and data caches characterized by 8 KB, 128 bit width and 4 words per line; iii) AXI bus with round-robin arbitration and 32 bit width; iv) Authentication Controller; and v) k -SUM PUF. The error correction is designed for a PUF with bit error probability of 15 % and a key error probability of 10^{-6} . Note that any other PUF can be used in SEPUFSoC.

Moreover, external 128 MB DDR and BRAM (integrated into the FPGA board) are used to store the application code/data and MACs, respectively. For the sake of high performance and security, different buses are used to connect the Authentication Controller with the off-chip memory and to the memory which stores the MACs. It improves the SEPUFSoC performance, by allowing simultaneously code/data and MAC accesses. AXI4-Lite protocol is employed to perform the read/write transactions. For expediting the repeatability, all IP cores used inside the SoC are included in Vivado 2015.3. The PUF is connected directly to the Authentication Controller. The CPU cores of the MPSoC share the same program and MAC memory. However, each CPU core has its own private authentication controller. A single PUF can be shared by both cores. Different keys are derived using different PUF responses and corresponding helper data.

To generate the 128-bit PUF-based key, a k -SUM PUF is built from 64 Ring Oscillator (RO) pairs, where the actual PUF implementation was not in the focus of this research. The module can be triggered to generate a new key and public data for key reconstruction or to regenerate a key from this stored data and the PUF. To reduce performance overhead through the PUF, the regeneration of a key is done only once after reboot. We assume that 50 % of the MPSoC traffic has security requirements and must be protected through the 64-bit MACs. Data gathered from the execution of the benchmark on the SoC/MPSoC is transmitted to the PC through the UART interface (AXI Uartlite). In our implementation, a MAC per cache line is computed (PTAG). The fine grained security offers flexibility in the data management, but increases the performance overhead. Further exploration in the MAC granularity computation is part of future work.

6.1 SEPUFSoC cost

The summary of the SEPUFSoC overhead when compared to the baseline system is presented in Table 2. Results show the reasonable low overhead of SEPUFSoC. Area and power results of SEPUFSoC Authentication Controller

Table 2: FPGA Resources utilization.

Component	FFs	LUTs	DSPs	BRAM
Baseline System	9421	10919	6	75
SEPUFSoc (Total)	12523	14800	6	76
Authentication Controller	1796	1934	0	0
PUF	1306	1947	0	1

Table 3: SEPUFSoc Authentication Controller – 65 nm.

Component	Area (um ²)	Leakage Power (nW)	Dynamic Power (nW)
Authentication Controller (Total)	27797	558	4624079
SipHash	19264	261	2506971
SipRound	8533	102	905346

are obtained with the Cadence Encounter RTL Compiler RC12.24 tool. The synthesis targets the 65 nm process technology at 100 MHz operating frequency and 25 °C. Results are shown in Table 3.

6.2 SEPUFSoc performance

SEPUFSoc memory access latency is higher than in the baseline implementation (without security) due to the additional computations performed by the authentication controller. Fig. 4 shows the timing diagrams for each SEPUFSoc access: cache line read, write and word write. The MAC computation, bottleneck of SEPUFSoc, requires 14 cycles: 10 cycles for the SipHash computation (a SipRound is executed each cycle) and 4 additional cycles due to the AXI4 bus protocol. Cache line read and write just require a single MAC computation. In contrast, the word write transaction is performed into 3 steps: i) fetches and verifies the integrity of the target memory line; ii) computes the new MAC; and iii) writes the data back. To evaluate SEPUFSoc performance, eight different benchmarks were employed: CoreMark, Dhrystone and a six MiBench applications (Basicmath, Bitcount, Qsort, Dijkstra, FFT, Stringsearch). Such benchmarks are widely used in the SoC/MPSoC evaluation [19]. They were compiled using the Xilinx Microblaze GNU toolchain in Xilinx SDK 2015.3 with the compiler optimization -O3. Fig. 5 shows the normalized results for the different benchmarks for different first-level cache memory sizes.

Table 4 presents the performance results of SEPUFSoc and the baseline MPSoC (without security). Results are expressed as CoreMark iterations per second (CM), Dhryst-

Figure 4: Performance figures of SEPUFSoc single transactions.

Figure 5: SEPUFSoc performance results.

Table 4: Summary of Performance Evaluation.

Benchmark	Units	Baseline	SEPUFSoc	Degradation
CoreMark	CM	186.1	177.8	4.4 %
Dhrystone	DMIPS	140.8	140.8	0 %
Basicmath	I/sec	0.4	0.3	12.5 %
Bitcount	I/sec	18.5	18.5	0.1 %
Dijkstra	I/sec	0.2	0.2	0.2 %
FFT	I/sec	1.5	1.5	0.2 %
Qsort	I/sec	273.9	204.2	25.4 %
Stringsearch	I/sec	39.3	38.6	1.6 %

tone Million Instructions per Second (DMIPS) and instructions per second (I/sec). Higher scores indicates a better system performance. Results show in general a small degradation of the performance due to the additional features of SEPUFSoc. These values depend on the number of byte writes of each benchmark. For instance Basicmath and Qsort demand a huge amount of byte writes, thus, forcing SEPUFSoc to constantly update the MACs.

6.3 Security requirements of SEPUFSoc

The main goal of the SEPUFSoc is to prevent illegitimate and unauthorized modifications of code/data applications. SEPUFSoc is able to successfully prevent spoofing

Table 5: Comparison of SEPUFSOC and the related works.

System	Security	Performance Penalty	Area Overhead (um ² , 65 nm)	Platform
CETD	Authenticated encryption	190.97 %	309971	Simulation
Yan et al.	Authenticated encryption	586.80 %	508424	Simulation
Rogers et al.	Authenticated encryption	510.21 %	582367	Simulation
SEPP	Encryption	50 %	–	FPGA Implementation
AEGIS	Encryption	73 %	–	FPGA Implementation
SEPUFSoc	Authentication	6.76 %–3.19 %	27797	FPGA Implementation

and relocation attacks, due to the inclusion of memory address and data for the MAC computation.

Security of SEPUFSoc strongly depends on the security of the PUF, i.e. its type, implementation, and error correction codes. To ensure a high level of security, there must not exist correlations between the derived secrets for different keys. Provided such a reasonable choice for the error correction, the secret generated from the PUF can be considered secure as long as the number of used bits does not exceed the entropy which can be provided by the PUF implementation. Note that the machine learning attacks against challenge-response authentication protocols based on so-called strong PUFs do not apply to the single challenge PUFs considered here.

Table 5 summarizes the results of SEPUFSoc and presents the comparison with previous works. SEPUFSoc is an efficient and low cost alternative for protecting memory content through authentication security service.

7 Conclusion

In this work we discussed the use of PUFs to achieve memory integrity and authentication in Multi-Processor System-on-Chip. We propose a PUF-based authentication and integrity verification architecture, called SEPUFSoc, to avoid code injection in SoC/MPSoc. The novelty of SEPUFSoc is the easy integration to general purpose CPUs and the protection of the different applications in MPSoc. We present the security requirements and the design exploration space of SEPUFSoc. We show that the security and efficiency of SEPUFSoc relies in the PUF and hash selection. The research on the area of emerging and new technologies for efficient and effective PUFs may led to new alternatives for efficient and secure PUFs able to be integrated into MSPoc. SEPUFSoc was implemented and evaluated under several MPSoc benchmarks. We show that SEPUFSoc is a flexible structure, which present good security and low impact on the MPSoc performance. As future work, we aim to implement different PUFs and hash

techniques. As well as to further increase the security of our system against replay attacks.

Funding: This work was partly funded by the German Federal Ministry of Education and Research (BMBF), grant number 01IS160253 (ARAMIS II) and the Fraunhofer High Performance Center for Secure Connected Systems of Munich.

References

1. A. B. Achballah et al. FW_IP: A flexible and lightweight hardware firewall for NoC-based systems. In *2018 International Conference on Advanced Systems and Electric Technologies (IC_ASET)*, 2018.
2. A. Adelsbach, U. Huber, and A.-R. Sadeghi. *Secure Software Delivery and Installation in Embedded Systems*, pages 27–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
3. D. Arumí, S. Manich, R. Rodríguez-Montañés, and M. Pehl. rram based random bit generation for hardware security applications. In *2016 Conference on Design of Circuits and Integrated Systems (DCIS)*.
4. J.-P. Aumasson and D. J. Bernstein. *SipHash: A Fast Short-Input PRF*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
5. G. T. Becker, A. Wild, and T. Güneysu. Security analysis of index-based syndrome coding for puf-based key generation. In *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, IEEE, 2015.
6. S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, Aug 2014.
7. J. Delvaux, D. Gu, D. Schellekens, and I. Verbauwhede. Helper data algorithms for puf-based key generation: Overview and analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(6):889–902, June 2015.
8. J. P. Diguët et al. Noc-centric security of reconfigurable soc. In *First International Symposium on Networks-on-Chip (NOCs'07)*.
9. S. Evain et al. From NoC security analysis to design solutions. In *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005*, pages 166–171, 2005.
10. R. Fernandes et al. A non-intrusive and reconfigurable access control to secure NoCs. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 316–319, 2015.

11. L. Fiorin et al. Security Aspects in Networks-on-Chips: Overview and Proposals for Secure Implementations. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 539–542, 2007.
12. L. Fiorin et al. A Security Monitoring Service for NoCs. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '08*, pages 197–202, ACM, New York, NY, USA, 2008.
13. L. Fiorin et al. Implementation of a reconfigurable data protection module for NoC-based MPSoCs. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.
14. C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing, STC '12*, ACM, New York, NY, USA, 2012.
15. C. G. Chaves, S. Payandeh Azad, T. Hollstein, and J. Sepúlveda. A distributed dos detection scheme for noc-based mpsoCs. pages 1–6, 10 2018.
16. F. Ganji, S. Tajik, and J.-P. Seifert. Why Attackers Win: On the Learnability of XOR Arbiter PUFs. In *Trust and Trustworthy Computing*, pages 22–39. Springer, 2015.
17. F. Ganji, S. Tajik, and J.-P. Seifert. Let me prove it to you: Ro pufs are provably learnable. In S. Kwon and A. Yun, editors, *Information Security and Cryptology – ICISC 2015*, pages 345–358, Springer International Publishing, Cham, 2016.
18. M. D. Grammatikakis et al. Security in MPSoCs: A NoC Firewall and an Evaluation Framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1344–1357, 2015.
19. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, Dec 2001.
20. C. Herder, M. Yu, F. Koushanfar, and S. Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, Aug 2014.
21. M. Hiller, M.-D. M. Yu, and G. Sigl. Cherry-Picking Reliable PUF Bits with Differential Sequence Coding. In *IEEE Trans. Inf. Forensics Security*, IEEE, 2016.
22. M. Hiller and A. G. Önalán. Hiding secrecy leakage in leaky helper data. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017: 19th International Conference, Taipei, Taiwan, September 25–28, 2017, Proceedings*, pages 601–619. Springer International Publishing, Cham, Sep 2017.
23. C. Hoffman, M. Cortes, D. F. Aranha, and G. Araujo. Computer security by hardware-intrinsic authentication. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2015.
24. Y. Hu et al. Automatic ILP-based Firewall Insertion for Secure Application-Specific Networks-on-Chip. In *2015 Ninth International Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip*, pages 9–12, 2015.
25. A. Juels and M. Wattenberg. A Fuzzy Commitment Scheme. In *Proceedings of the 6th ACM Conference on Computer and Communications Security, CCS '99*, pages 28–36. ACM, 1999.
26. S. Kleber, F. Unterstein, M. Matousek, F. Kargl, F. Slomka, and M. Hiller. Secure execution architecture based on puf-driven instruction level code encryption. *Cryptology ePrint Archive*, Report 2015/651, 2015. <http://eprint.iacr.org/2015/651>.
27. G. Kornaros, O. Tomoutzoglou, and M. Coppola. Hardware-Assisted Security in Electronic Control Units: Secure Automotive Communications by Utilizing One-Time-Programmable Network on Chip and Firewalls. *IEEE Micro*, 38(5):63–74, Sep 2018.
28. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *SIGPLAN Not.*, 35(11), Nov. 2000.
29. D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. *SIGOPS Oper. Syst. Rev.*, 37(5), Oct. 2003.
30. R. Maes, A. Van Herreweghe, and I. Verbauwhede. PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 302–319. Springer, 2012.
31. A. Malekpour, R. Ragel, A. Ignjatovic, and S. Parameswaran. Dosguard: Protecting pipelined mpsoCs against hardware trojan based dos attacks. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 45–52, July 2017.
32. E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, ACM, New York, NY, USA, 2013.
33. M. Pehl, M. Hiller, and G. Sigl. Secret Key Generation and Authentication. In *Information Theoretic Security and Privacy of Information Systems*, Cambridge University Press, 2017.
34. J. Sepúlveda et al. Dynamic NoC-based Architecture for MPSoC Security Implementation. In *Proceedings of the 24th Symposium on Integrated Circuits and Systems Design, SBCCI '11*, pages 197–202, ACM, New York, NY, USA, 2011.
35. J. Sepúlveda, M. Gross, A. Zankl, and G. Sigl. Exploiting Bus Communication to Improve Cache Attacks on Systems-on-Chips. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '17)*, July 2017.
36. J. Sepúlveda, R. Pires, G. Gogniat, W. J. Chau, and M. Strum. Qoss hierarchical noc-based architecture for mpsoC dynamic protection. *International Journal of Reconfigurable Computing*, 2012:3, 2012.
37. J. Sepúlveda, F. Willgerodt, and 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*, pages 39–44, ACM, New York, NY, USA, 2018.
38. J. Sepúlveda, A. Zankl, D. Flórez, and G. Sigl. Towards protected mpsoC communication for information protection against a malicious noc. *Procedia Computer Science*, 108:1103–1112, 2017. International Conference on Computational Science, ICCS 2017, 12–14 June 2017, Zurich, Switzerland.
39. J. Sepúlveda, D. Flórez, and G. Gogniat. Reconfigurable security architecture for disrupted protection zones in noc-based mpsoCs. In *2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8, June 2015.

40. M. J. Sepúlveda, J. Diguët, M. Strum, and G. Gogniat. Noc-based protection for soc time-driven attacks. *IEEE Embedded Systems Letters*, 7(1):7–10, March 2015.
41. S. P. Skorobogatov. *Semi-invasive attacks: a new approach to hardware security analysis*. PhD thesis, University of Cambridge, 2005.
42. G. E. Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, Massachusetts Institute of Technology, Aug 2005.
43. 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, ICS '03*, ACM, New York, NY, USA, 2003.
44. G. E. Suh, C. W. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *IEEE Design Test of Computers*, 24(6), Nov 2007.
45. 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 *32nd International Symposium on Computer Architecture (ISCA'05)*, June 2005.
46. M. Weiner, S. Manich, R. Rodríguez-Montañés, and G. Sigl. The low area probing detector as a countermeasure against invasive attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(2):392–403, Feb 2018.
47. F. Wilde. Large scale characterization of sram on infineon xmc microcontrollers as puf. In *4th Workshop on Cryptography and Security in Computing Systems (CS2 2017) HIPEAC17*, Stockholm, Sweden, Jan 2017.
48. F. Wilde, B. M. Gammel, and M. Pehl. Spatial correlation analysis on physical unclonable functions. *IEEE Transactions on Information Forensics and Security*, 13(6):1468–1480, June 2018.
49. M.-D. M. Yu. Recombination of physical unclonable functions. In *GOMACTech-10 Conference*, 2010.

Bionotes

Johanna Sepúlveda
Technical University of Munich, Munich,
Germany
johanna.sepulveda@tum.de

Johanna Sepúlveda received the M. Sc. and Ph. D. degrees in Electrical Engineering – Microelectronics by the University of São Paulo, Brazil in 2006 and 2011, respectively. She was Postdoctoral fellow at the Integrated Systems and Embedded Software group at this Uni-

versity and at the group of Embedded Security of the University of South Brittany, France. Moreover, Dr. Sepúlveda was a Visiting Researcher at the Computer Architecture group at the University of Bremen, Germany. In 2014, she worked as a Senior INRIA Postdoctoral researcher at the Heterogeneous Systems group at the University of Lyon, France. Since 2015, she holds a Senior Researcher Assistant position at the Technical University of Munich, Germany. She has been working in the field of embedded security design for more than 10 years. Her research interest also includes high performance SoC design and new technologies design.

Felix Wilgerodt

Technical University of Munich, Munich, Germany
wil.Ger38@net.de

Felix Wilgerodt received his Master degree in electrical engineering and information technology from the Technical University of Munich. During his master's thesis he worked on the integration of security at Multi-Processors System-on-Chip.

Michael Pehl
Technical University of Munich, Munich,
Germany
m.pehl@tum.de

Michael Pehl was born in Munich, Germany in 1978. He received the Dipl.-Ing. degree in electrical engineering and information technology and the Dr.-Ing. Degree (summa cum laude) from the Technical University of Munich in 2006 and 2012, respectively. Since 2012, he has been a researcher with the Chair for Security in Information Technology, Department of Electrical and Computer Engineering, Technical University of Munich, where he is leading the Physical Unclonable Functions Group. His research has been concerned with tools for design automation of analog circuits as well as with hardware security, with a focus on different aspects of Physical Unclonable Functions (PUFs), such as evaluation of PUFs, error correction for PUFs, side-channel attacks on PUFs, and applications of PUFs.