A Novel FPGA Architecture and Protocol for the Self-attestation of Configurable Hardware

Jo Vliegen*, Md Masoom Rabbani**, Mauro Conti**, *Senior Member, IEEE*,
Nele Mentens*, *Senior Member, IEEE*

*ES&S and imec-COSIC/ESAT, KU Leuven, Belgium, Email: firstname.lastname@kuleuven.be
**SPRITZ, University of Padua, Italy, Email: lastname@math.unipd.it

Abstract—Field-Programmable Gate Arrays or FPGAs are popular platforms for hardware-based attestation. They offer protection against physical and remote attacks by verifying if an embedded processor is running the intended application code. However, since FPGAs are configurable after deployment (thus not tamper-resistant), they are susceptible to attacks, just like microprocessors. Therefore, attesting an electronic system that uses an FPGA should be done by verifying the status of both the software and the hardware, without the availability of a dedicated tamper-resistant hardware module.
Inspired by the work of Perito and Tsudik, this paper proposes a partially reconfigurable FPGA architecture and attestation protocol that enable the self-attestation of the FPGA. Through the use of our solution, the FPGA can be used as a trusted hardware module to perform hardware-based attestation of a processor. This way, an entire hardware/software system can be protected against malicious code updates.

Index Terms—FPGA, configurable hardware, remote attestation, hardware-based attestation, partial reconfiguration, ICAP, configuration readback

1 Introduction

Field-Programmable Gate Arrays (FPGAs) combine the flexibility of software with the performance of hardware: they allow device reconfiguration in the field while offering a higher performance per consumed energy unit than general-purpose microprocessors. In comparison to Application-Specific Integrated Circuits (ASICs), FPGA applications have a shorter time to market and can be designed with a lower non-recurring engineering (NRE) cost. ASICs are not configurable after deployment but lead to circuits with a higher speed, a lower power consumption and a smaller area than FPGAs. Nevertheless, the performance gap between FPGAs and ASICs is continuously shrinking thanks to two phenomena: (1) the high-volume production of FPGAs makes it economical to closely follow the latest technology nodes, and (2) FPGA vendors improve the performance of FPGAs by integrating dedicated application-specific building blocks. These evolutions make the use of FPGAs in embedded systems increasingly popular.

A typical FPGA-based embedded system combines a general-purpose microprocessor with configurable hardware. For the microprocessor, several techniques have been proposed to verify that it is running the intended software application, as explained in Section 4. However, for the FPGA, it is not straightforward to remotely verify that it is configured to the intended state. Many attestation mechanisms for microprocessors rely on a tamper-resistant hardware module. Assuming that the hardware module itself can be remotely reconfigured, the hardware prover core needs to be able to prove its own state to the verifier, i.e., the configurable hardware needs to perform self-attestation. This is shown in Figure 1, where the microprocessor and the tamper-resistant hardware module are denoted by µP and TR HW, respectively. The left side of the figure shows the traditional adversary model, in which the adversary is assumed to be capable of changing the software code in the processor. The right side of the figure shows the scenario that is considered in this work, where the adversary can additionally tamper with the FPGA configuration.

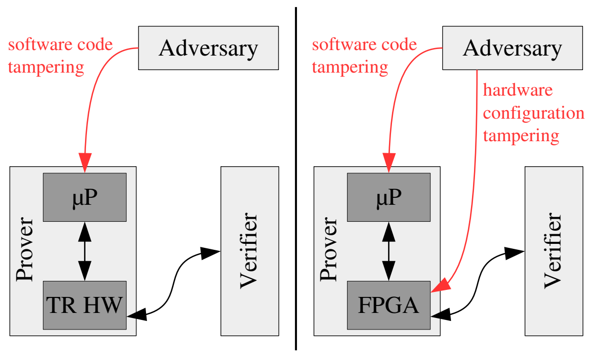


Fig. 1: The role of the adversary in traditional hardware-based attestation (left) and in the model considered in this paper (right), where µP denotes the microprocessor and TR HW stands for the tamper-resistant hardware module.

The mechanism that is proposed, is inspired by the work of Perito and Tsudik [1], who apply proofs of secure erasure and secure code updates to embedded processors. They assume that the processor platform contains a small amount of immutable read-only memory (ROM) that stores a basic program, taking care of communication and memory read/write. FPGAs, however, do not have the possibility of directly storing and accessing their basic program/functionality in an immutable piece of ROM. Since the basic functionality is stored in configurable memory, it is far from straightforward to apply the results of [1] directly to FPGAs.

This paper presents the SACHa (Self-Attestation of Configurable Hardware) scheme. It consists of a novel hardware design, mapped on an off-the-shelf FPGA, and a communication protocol that executes the attestation process based on the proposed FPGA design. The paper extends our preliminary work [2] in two ways: (1) it dives deeper into the technological capabilities of FPGAs that are needed to implement the SACHa scheme, and (2) it details the practical implementation and measurement results of the scheme on a Xilinx FPGA. This allows other researchers and practitioners to repeat our experiments and build further on the SACHa scheme.

The paper is structured as follows. First, Section 2 gives some background information. Section 3 discusses the assumed system model and adversary model. In Section 4, we revisit the concepts of remote attestation by discussing related work. Section 5 introduces the SACHa scheme and architecture and Section 6 presents a proof-of-concept implementation. The performance and security of SACHa are evaluated in Section 7. Finally, Section 8 concludes the paper and gives directives for future work.

2 Preliminaries

This section starts with explaining the basic structure of an FPGA. Subsequently, it elaborates on two specific features that we use in our SACHa proposal, namely partial reconfiguration and configuration memory readback. Finally, the concept of attestation is introduced as well as the specific attestation solution that lies at the basis of this work.

2.1 FPGA

2.1.1 Basic Structure

The FPGA has been around for more than thirty years. It consists of configurable fabric which gets its configuration from a configuration memory, as shown on the left side of Figure 2. Through this configuration memory, the functionality of the configurable fabric is determined. The data that are stored in the configuration memory are referred to as the bitstream. Depending on the type of FPGA, the configuration memory can be (volatile) SRAM or (non-volatile) Flash memory. This work focuses on SRAM-based FPGAs, which is the most frequently applied type of FPGAs. More specifically, in the remainder of this paper, we concentrate on Xilinx FPGAs and use the corresponding terminology. Nevertheless, the concepts we propose can be applied to most SRAM-based FPGAs.

The basic building blocks of the configurable fabric are Configurable Logic Blocks (CLBs), embedded memory blocks called Block RAMs (BRAMs), Input/Output Blocks (IOBs) and Switch Matrices (SMs); as is shown on the right side of Figure 2. The CLBs consist of look-up tables and distributed storage elements, while the BRAMs provide centralized memory. The actual functionality of the FPGA design is configured on the CLBs and the BRAMs. The SMs interconnect the CLBs and the BRAM to each other and to the IOBs. And they also connect the internal hardware to the external environment through the pins of the FPGA. All of the mentioned elements are configured by the bits in the configuration memory.

Note that FPGAs also contain other dedicated hardware primitives, which we omit from this overview, since they are not necessary for the implementation of our solution. Nevertheless, it is possible to use these primitives in combination with the proposed FPGA architecture.

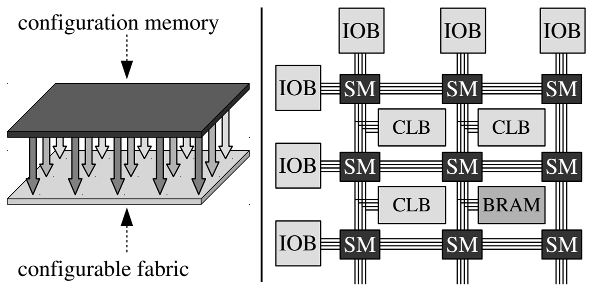


Fig. 2: Conceptual representation of an FPGA (left) and basic building blocks of the configurable fabric (right).

2.1.2 Partial Reconfiguration

An FPGA can be logically partitioned, which implies that the configurable fabric is segmented in two or more partitions. These partitions can be configured separately while the other partitions continue to operate normally. The part of the configuration memory that configures a specific partition is then updated at run-time through a bitstream of which the size is proportional to the size of the partition. This is referred to as partial reconfiguration.

The configuration memory of Xilinx FPGAs is not only accessible from the outside of the FPGA, but also from the configurable fabric inside the FPGA. This is achieved through a dedicated primitive called the Internal Configuration Access Port (ICAP). When dealing with multiple partitions, one partition usually stays unchanged and contains the ICAP together with control logic. This partition is referred to as the static partition. Typically, the configuration of the static partition is loaded from a non-volatile Flash memory on the printed circuit board into the (volatile) SRAM-based configuration memory when the power is turned on.

Next to the static partition, there can be one or more run-time configurable partitions, which are referred to as reconfigurable or dynamic partitions. This is shown in Figure 3, in which the ICAP is used to write a bitstream into the part of the configuration memory that is connected to the dynamic partition. This results in the reconfiguration of the dynamic partition. Such a bitstream, only targeting a dynamic partition, is referred to as a partial bitstream. Note that, in principle, the ICAP is capable of updating the entire configuration memory, including the static partition. Nevertheless, this setting is rarely used in practice, because the control logic in the static partition that interacts with the ICAP should not be changed.

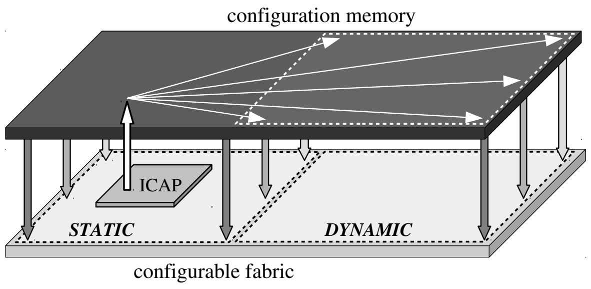


Fig. 3: FPGA design in which the ICAP in the static partition updates the configuration of the dynamic partition.

2.1.3 Configuration Memory Readback

Considering applications in which (un)intended faults occur in the configuration memory, the readback capabilities of the ICAP can be used for error detection and correction. This is important in e.g., space applications, in which Single Event Upsets cause bit flips in the configuration memory. We do not target the detection of faults in our solution, but rather the presence of malicious configuration data. Therefore, we also use the configuration memory readback mechanism, which allows the ICAP to read out the entire configuration memory, as shown in Figure 4.

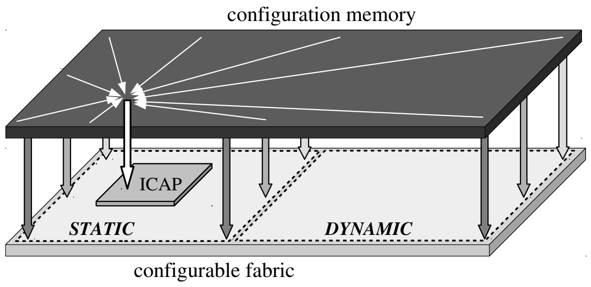


Fig. 4: FPGA design in which the ICAP in the static partition reads back the configuration of the entire configuration memory.

2.2 Attestation Concept

In general, attestation is a challenge-response protocol between a verifier and an untrusted prover. Through attestation, the verifier determines the "health" of the prover. In a typical attestation protocol, the prover sends a cryptographic checksum of its current state upon request of the verifier. Based on the received checksum, the verifier determines if the prover is operating in the intended state. In order to ensure the freshness of the response, a nonce generated by the verifier is included in the checksum. This is shown in Figure 5.

The attestation mechanism we use in this paper relies on proofs of secure erasure, which ensure that the memory/state of an embedded device is erased. This way secure code updates can be done to ensure that the memory/state of an embedded device is updated. It takes advantage of the bounded memory model of an embedded device, which assumes the verifier knows the exact size of the prover's

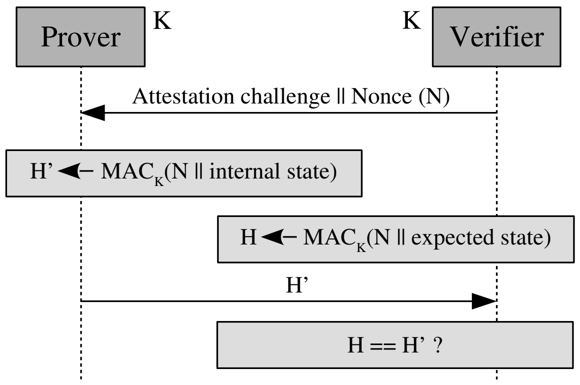

Prover
K
K
Verifier
Attestation challenge || Nonce (N)
H' ◄— MAC_K(N || internal state)
H ◄— MAC_K(N || expected state)
H'
H == H' ?

Fig. 5: Typical example of an attestation protocol between a verifier and a prover, using a key K.

(bounded/limited) memory. The original proposal, as introduced by Perito and Tsudik in [1], by which our work is inspired, can be summarized as follows. When the verifier sends data or code to the prover that fills the entire (limited) memory of the prover's embedded device, it is implied that all prior code is overwritten and thus erased. The device can then compute the checksum of the memory content and send it back to the verifier. The embedded device is supposed to have a small amount of immutable ROM that takes care of (1) receiving code updates and writing them to the device's memory, and (2) reading out the checksum and sending it back to the verifier. The algorithm for the computation of the checksum can either be included in the code that is sent by the verifier as part of the protocol, or it can be a (fixed) part of the immutable ROM. When we apply the mechanism proposed in [1] to Figure 5, the attestation challenge and the nonce correspond to the code that is sent by the verifier to fill the entire memory of the prover's device. The Message Authentication Code (MAC) corresponds to the cryptographic checksum of the whole memory content. This way, the goal is not to detect the presence of malicious code, but to make sure there is no malicious code remaining after the code erasure/update.

We apply a similar concept to FPGAs. In order to do so, we overcome the challenges that occur due to the differences between embedded processor platforms and FPGAs. The resulting FPGA architecture uses partial reconfiguration and configuration memory readback to make sure that it does not contain malicious hardware modules. This way, the FPGA can perform self-attestation, which is crucial for hardware-based attestation solutions that use an FPGA as the trusted hardware module.

3 System and Adversary Model

Table 1 lists the notation we use for the entities in the attestation scheme and the components of the system on the prover's side. The system model, consisting of the entities and components in the table, is depicted in Figure 6.

The system model consists of the *Prv* and *Vrf* who communicate with each other over a public channel. The *Vrf* is not constrained in computing power and is typically a laptop, a desktop computer or a server. The *Prv* is an

TABLE 1: Notations used in this work.

Entities	
Prv, *Vrf*	prover and verifier, respectively
Adv	adversary
Components of the *Prv*	
StatPart	static partition of the *Prv*'s FPGA
DynPart	dynamic partition of the *Prv*'s FPGA
StatMem	configuration memory for *StatPart*
DynMem	configuration memory for *DynPart*
BootMem	non-volatile read-only memory, external to the FPGA, to boot *StatMem*

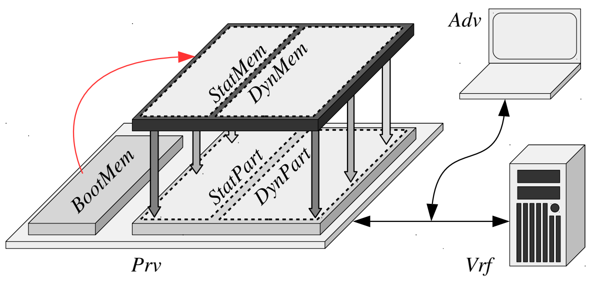


Fig. 6: System model.

embedded system that consists of an FPGA and a *BootMem*, that initializes the *StatMem* when the power is turned on. We assume that the *BootMem* is programmed before deployment and not accessible remotely. Therefore, it can be seen as a read-only memory during the attestation process. Note that the *BootMem* is not the same as the immutable ROM in the processor-based solution of Perito and Tsudik [1], because it is not directly used to determine the functionality of the FPGA; its content is loaded into the *StatMem* at power-on, but it is the content of the *StatMem*, which is not immutable, that determines the functionality of the *StatPart*. The *DynMem* can be repeatedly reconfigured after start-up. The *StatMem* and the *DynMem* provide the configuration for the *StatPart* and the *DynPart*, respectively.

The adversary model depicts the scenario where the *Adv* compromises or impersonates the *Prv* to fake its current state or behavior to the *Vrf*. In most of the attestation literature, software-only attackers are considered. In the scenario that we consider, a processor running software is connected to an FPGA-based trusted component. Our *Adv* can modify both the software of the processor and the hardware configuration of the FPGA, i.e. the data in the configuration memory. In this paper, we concentrate only on the attestation of the FPGA configuration. We assume that the *Adv* is capable of modifying the configuration memory of the FPGA, not of applying hardware modifications to the configurable fabric of the FPGA. This also excludes Hardware Trojan insertion from the attack space. We can classify our *Adv* based on the taxonomy introduced in [3]:

- The *Adv* can be a "remote adversary" that aims at inserting malicious hardware components on the *Prv* remotely. An example of an attack performed by a remote adversary is the 2010 Stuxnet incident [4].
- The *Adv* can be a "local adversary" (subsuming a remote adversary) that aims at impersonating or cloning the *Prv*'s device and/or at collecting information. The *Adv* does this by eavesdropping and/or controlling the communication between *Prv* and *Vrf*.

We consider side-channel analysis attacks and physical attacks that actively modify the configurable fabric or the FPGA-based system out of our current scope.

4 Related Work

We explore related work in remote attestation in this section. The discussed methods mainly belong to either software-based or hard-ware-based attestation. Apart from that, we also consider hybrid techniques which employ minimum hardware support.

4.1 Software-based Attestation

In general, most of the software-based attestation mechanisms do not require hardware support and rely on a challenge-response protocol. Typically, in software-based attestation methods, a *Vrf* sends a challenge to a *Prv* (device). The *Prv* computes the cryptographic checksum of its own memory or underlying software along with the challenge provided by the *Vrf* and sends it back to the *Vrf*. Based on the received response, the *Vrf* verifies the "state" of the *Prv*.

In [5], Spinellis et al. propose a mechanism in which the *Prv* computes the hash of two randomly colluding memory areas. The hash value is then sent to the *Vrf*, who compares it to the expected hash values. This technique relies on sequential memory read-out for the hash calculation, but the data memory is not verified. In case of an intelligent adversary, malicious code can evade detection by shifting its locations; this flaw occurs due to the non-simultaneous hash calculation of the two randomly overlapping areas.

Seshadri et al. propose a software-based attestation scheme called SWATT [6]. It assumes that malicious code that is running on a (compromised) *Prv* must re-direct the memory access to the location where the actual code resides in order to get the valid response for the attestation challenge. The authors assume that the timing overhead introduced by the memory re-direction will be noticed during the protocol execution. SWATT relies on strict timing constraints, and thus unfeasible for real-world employment over a network.

A remote software-based attestation scheme to detect a malicious *Prv* in a network is proposed by Shaneck et al. in [7]. The attestation challenge is generated at run-time and is shared with the *Prv* using symmetric-key encryption to achieve secure communication. A vulnerability occurs when the node is compromised and the shared symmetric key is extracted. The authors also use self-modifying code to prevent an adversary from evading detection. However, this technique does not verify the data memory and an intelligent adversary can still evade detection by relocating its position during attestation.

In other software-based attestation schemes like the one proposed by Choi et al. [8], the *Prv*'s memory is filled by pseudo-randomness using a Pseudo Random Function (PRF). The *Vrf* sends a nonce to the *Prv*, after which the *Prv* uses the nonce as a seed for the PRF. The value generated by the PRF then fills the empty memory region of the *Prv*.

Next, the *Prv* computes the hash of the memory and sends the result to the *Vrf* for verification. The main idea is to fill the empty memory regions in such a way that the adversary will have no place to hide malicious code. However, a compromised *Prv* having access to the PRF can still evade detection by computing a valid hash.

Li et al. present a technique for verifying the integrity of peripherals' firmware (VIPER [9]), which is also a software-based attestation technique to identify the presence of malware in the firmware of the peripherals. VIPER is a challenge-response protocol that runs between a host CPU (*Vrf*) and a potentially untrusted peripheral (*Prv*). The idea is to identify the presence of malware which tries to hide itself by employing a more powerful and faster proxy to respond to the challenge sent by the *Vrf*. Unfortunately, VIPER also depends on a strict time-bound response and does not scale as the *Vrf* has to check the peripherals one at a time, thus making it impractical for large-scale industrial implementation.

In summary, software-based attestation schemes are interesting, thanks to their easy and low-cost "hardware-less" approach. However, most of the schemes have flaws or are not practical due to strict timing constraints, due to the absence of data memory attestation and/or due to the lack of protection of stored secrets when the node is compromised.

4.2 Hybrid Attestation

Hybrid attestation schemes employ software/hardware co-design that facilitates effective, low-cost, secure solutions without a dedicated hardware module (e.g., a TPM) to thwart the inefficiency of software based-attestation schemes. The goal of hybrid architectures is to provide more security to the attestation schemes against all adversaries except for physical adversaries.

In [10], El Defrawy et al. propose SMART (Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust), a software/hardware co-design for low-end embedded devices. The essence of this architecture is to provide a secure memory location for attestation code and for an attestation key. The processor, to which minimal changes in the form of these secure memory locations have been applied, protects the secure memory locations from "non-SMART" codes.

With TrustLite, Koeberl et al. provide a seclusion of specific software modules, independent of the operating system, known as Trustlets [11]. They introduce an Execution-Aware Memory Protection Unit (EA-MPU), which has a similar working principal as the SMART-based memory protection unit. The EA-MPU enforces code-specific data use. Ferdinand et al. propose Tiny Trust Anchor for Tiny Devices (TyTAN) [12]. The core idea of this architecture is based on an EA-MPU. Apart from providing secure inter-process communication, TyTAN facilitates robust scheduling and run-time loading and unloading of tasks.

More recently, in [13]–[15], the authors propose new hybrid schemes for low-end devices to counter physical adversaries. Especially in [13], the use of a reliable read-only clock guarantees non-malleability of the attestation results. Specialized secure hardware modules, e.g. a Memory Protection Unit, safeguard the attestation-related code and keys from unauthorized access. These kind of secure tamper-resistant hardware units assure authenticity of the attestation results. However, the main purpose of these techniques is to identify the presence of physical adversaries rather than protecting the device itself.

The aforementioned schemes are designed while keeping in mind low-end, tiny devices. Apart from providing better resilience against stronger adversaries in networks, their development and deployment in low-end devices make large-scale "swarm" attestation feasible, i.e., a number of low-end, tiny embedded devices that are employed as a group for a specific task.

4.3 Hardware-based Attestation

Hardware-based attestation methods predominantly rely on the use of specialized hardware. Arbaugh et al. propose the "AEGIS" architecture to ensure the integrity of the *Prv* [16]. The essence of this method is a list of security checks on the BIOS that are done from power-on until the kernel is loaded. Failure of any of these checks will reboot the *Prv* and bring it back to a known saved state.

In order to check the trustworthiness of the *Prv*, Sailer et al. propose to extend the Trusted Platform Module (TPM) with additional functionality [17]. The main idea is that the TPM maintains a sequence of trust which covers the application layer and the system configuration. Furthermore, a kernel-maintained checksum list is also included in the TPM for preserving its integrity.

In [18], England et al. propose to segregate a system into two parts, namely a trusted and an untrusted part. Both parts have distinct operating systems. Only the trusted part of the system will be checked to maintain the integrity of the system.

Kil et al. propose ReDAS (Remote Dynamic Attestation System) in [19]. Their approach consists of extracting the properties from application source code. At the time of program execution, all activities, including malicious activities, are recorded. The *Prv* is equipped with a TPM which stores the recorded values in order to protect them against adversarial modification. Upon receiving the attestation request (challenge) from the *Vrf*, the *Prv* sends the TPM-protected information to the *Vrf*. Although this approach is better than the other discussed approaches, it has a drawback: ReDAS does not consider all the available properties; it only checks a subset of the dynamic system properties. As a result, an adversary can still be successful by modifying properties which are not covered by ReDAS.

The aforementioned hardware-based attestation solutions rely on tamper-resistant hardware modules. These tamper-proof modules cannot be modified by a physical adversary, but are very costly and therefore unfeasible to deploy on low-cost, tiny devices. For higher-end, security-critical devices, hardware-based attestation is preferred over software-based attestation.

We are aware of two papers that deal with the remote attestation of configurable hardware; they are discussed in the following two paragraphs.

Drimer and Kuhn describe a protocol for secure remote updates of FPGAs in [20]. The presented protocol provides for the remote attestation of the running configuration and the status of the upload process. The bitstream is stored in an external non-volatile memory and the configuration memory is assumed to be tamper-proof. Other work on the secure remote configuration of FPGAs is described in [21], [22].

In [23], Chaves et al. perform on-the-fly attestation of configurable hardware. Given that a loadable hardware structure to a configurable device is described by a binary bitstream, the hash value of this bitstream is calculated to validate the hardware structure. The attestation core implemented in the FPGA is assumed tamper-proof, as the core is supposed to make sure that partial configuration updates can only take place in a predetermined restricted area.

Both [20] and [23] rely on an external memory that is accessible during the attestation process and/or a tamper-proof configuration memory. Our system and adversary model are much stronger, assuming that the attestation mechanism cannot rely on an external memory and assuming that the configuration memory is not tamper-proof. Therefore, our solution is the first mechanism to propose the self-attestation of an FPGA.

Note that our solution uses a PUF, as explained in the remainder of the paper. There are other attestation mechanisms that use PUFs. However, these mechanisms rely on a strong PUF to generate a challenge-based response, while our solution uses a weak PUF to generate a key. This means that the PUF has a less important role in our solution; the novelty of our work is in the dynamic reconfiguration based architecture in combination with the use of the bounded memory model. Therefore, we do not extensively compare to other PUF-based attestation mechanisms.

5 Our Solution: SACHa

5.1 Contribution

The mechanism we introduce in this paper improves the security of FPGA-based attestation methods. From the observation that the trusted hardware module itself needs to be verified when it is based on configurable hardware (i.e., an FPGA), we propose the SACHa architecture and attestation protocol. SACHa allows the self-attestation of the FPGA-based module, such that the FPGA can be trusted by the *Vrf* when it is used for the hardware-based attestation of the software running on a processor. In this paper, we concentrate on the self-attestation of the FPGA, not on the connection of the FPGA-based trusted module to a processor. Nevertheless, our solution can easily be combined with existing hardware-based attestation mechanisms. SACHa consists of a novel FPGA architecture (implemented on an off-the-shelf FPGA) and attestation protocol.

5.2 FPGA Architecture

We apply the bounded memory model, introduced in [1] and summarized in Section 2.2, to the configuration memory of an FPGA. We rely on the observation that the FPGA does not have enough memory in the configurable fabric to store the configuration data sent by the *Vrf*. In [24], it is shown that this is a realistic assumption, i.e. the internal BRAM does not have enough capacity to store a bitstream that configures a large part of the FPGA. Therefore we can be sure that the configuration data are stored in the configuration memory. Since the configuration data stored in the memory determine the functionality of the configurable fabric (cfr. Figure 2), this automatically implies that the configurable fabric of the FPGA is running the application intended by the *Vrf*. The platform used in [1] is assumed to have an immutable ROM that contains a program for basic send/receive and read/write functionality. Since FPGAs do not have this as a part of their configuration memory, we propose an architecture that makes use of partial reconfiguration. In our solution, the communication with the *Vrf* and the configuration memory read/write mechanism are implemented in the *StatPart*. The code updates are applied to the (bounded) *DynMem*. A cryptographic checksum is computed on the entire configuration memory, covering both the *StatMem* and the *DynMem*. Figure 7 gives a high-level overview of the FPGA architecture.

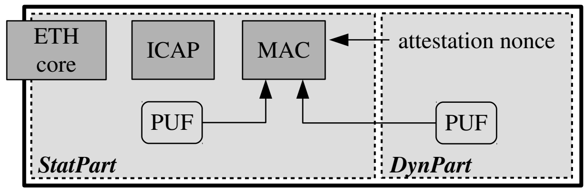


Fig. 7: High-level FPGA architecture of SACHa, in which the key for the MAC comes either from *StatPart* or from *DynPart*.

5.2.1 Static Partition

In the *StatPart*, the ICAP takes care of writing the configuration memory in order to (re)configure the *DynPart*. It is also used for reading out the entire configuration memory, which contains both *StatMem* and *DynMem*. Further, the Ethernet core (ETH core) provides a communication link with the *Vrf*. The MAC core computes the cryptographic checksum of the entire configuration memory content. The MAC serves two purposes: (1) it guarantees that the checksum is computed by the FPGA and not by another device impersonating the FPGA (this is achieved by a shared key between *Prv* and *Vrf*); (2) it guarantees that the configuration data are not tampered with.

There are two options for generating the key for the MAC in the device. The first option is to implement a (weak) Physical(ly) Unclonable Function (PUF) in the *StatPart* that generates the key. Even if the *Adv* has access to the PUF circuit in the *StatMem*, the key cannot be retrieved to clone the device. The second option is to include a PUF in the *DynPart* as a new hardware module from the *Vrf* as part of the attestation protocol. This allows the *Vrf* to update the shared key by updating the PUF circuit. In this case, each PUF circuit sent by the *Vrf* needs to have gone through an enrollment phase before the deployment of the FPGA. Further, the *Vrf* needs to keep a database of PUF circuits and corresponding keys. Note that we assume an ideal key-generating PUF in our solution; attacks/weaknesses of PUFs are considered out of scope in this work. The use of a PUF does lead to an additional enrollment step in the preparation of the FPGA-based device. However, the

BootMem of the device needs to be programmed anyway, so we assume that the enrollment and thus the key exchange can be done in the same provisioning step, which takes place before the devices are placed in the field.

The *StatPart* needs to be configured and running on the FPGA at all times. Since we focus on SRAM-based FPGAs, the configuration memory is volatile. This means that the static configuration needs to be loaded from a non-volatile memory every time the power of the FPGA is turned on. Therefore, we include *BootMem* in the system, i.e. a small Flash memory to load the *StatMem* of the FPGA at power-on. We minimize the size of the *BootMem*, such that it is not capable of storing the configuration bitstream of the *DynPart*, since that would undermine our assumption that the partial bitstream can only be stored in the configuration memory. In order to achieve a minimum-size static configuration bitstream and thus a minimum-size *BootMem*, we make the area of the *StatPart* as small as possible and for sure significantly smaller than the area of the *DynPart*. Note that, on commercial FPGA boards, it is only possible to program the *BootMem* by decoupling it from the board and connecting it to a programming device. This means that, even if the *BootMem* was capable of storing the full bitstream of the FPGA, it would still not be possible to store the partial bitstream sent remotely by the *Prv*. So we can safely assume that the bitstream sent by the *Vrf* can only be stored in the configuration memory.

5.2.2 Dynamic Partition

The *DynPart* contains the intended configuration of the FPGA and a register that stores a nonce, i.e., an arbitrary number that can only be used once. The nonce can be updated by the *Vrf* in order to achieve freshness when requesting a MAC from the *Prv*. Optionally, the *DynPart* contains a PUF for key generation, as explained in Section 5.2.1. In practice, we propose to use a separate partition for the nonce, such that the nonce can be updated without updating the intended application in the *DynPart*. This way, the *Vrf* can request a fresh checksum of the *Prv*'s configuration without changing the intended application. Note that the nonce could also be communicated to the *StatPart* as a normal data packet.

5.3 Attestation Protocol

Figure 8 shows the attestation protocol that is applied between *Vrf* and *Prv*, in which the SACHa FPGA architecture is on the side of the *Prv*. First, the *Vrf* sends a partial bitstream to the *Prv*, who stores the bitstream in the configuration memory through the ICAP. As explained in Section 5.2.2, the architecture facilitates the independent configuration of the intended application and the nonce. Therefore, the dynamic configuration consists of two steps, as shown in Figure 8. After the two configuration steps, the entire *DynMem* is (over)written by the *Vrf*. Note that, even if the intended application and the nonce register do not need all the resources in the configurable fabric of the dynamic partition, the partial bitstream still fills the entire *DynMem*. Optionally, the bitstream that configures the intended application also contains configuration data for the key-generating PUF.

When the bitstream is written into the configuration memory, the FPGA runs the intended application and stores

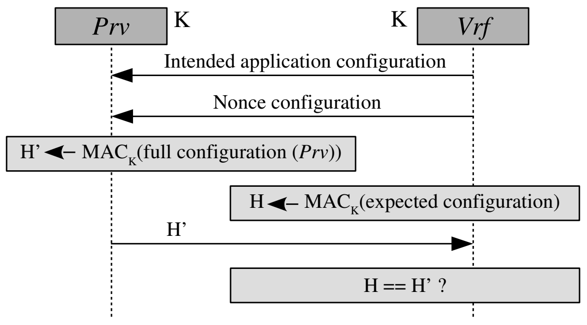

Prv
K
K
Vrf
Intended application configuration
Nonce configuration
H' ← MAC_K(full configuration (Prv))
H ← MAC_K(expected configuration)
H'
H == H' ?

Fig. 8: SACHa protocol, using a key K.

the received nonce. To prove this to the *Vrf*, the entire configuration memory is read out by the ICAP. A MAC is generated on the read-back data and sent back to the *Vrf*, who generates the same MAC using the shared key. Finally, it compares the two values to verify the internal configuration of the entire FPGA.

6 Proof-of-concept Implementation

As a proof of the SACHa concept, an implementation is made on a Xilinx Virtex 6 FPGA (XC6VLX240T). To generate configuration bitstreams, we use the Xilinx ISE 14.7 Suite. The implementation of the protocol and the architecture are discussed in this section.

6.1 Implementation of the Protocol

The configuration memory of the XC6VLX240T FPGA consists of 28,488 frames. A frame is the smallest addressable part of the configuration memory and contains 81 words of 32 bits for the considered FPGA. Since we want to make the *StatPart* as small as possible, we use a BRAM-based memory to store a single bitstream frame. This means that the *Vrf* sends a single frame per network packet until the *DynMem* of the FPGA is completely (over)written and the *DynPart* is completely (re)configured. A trade-off between the size of the BRAM-based memory and the number of communication steps can be made, as long as the memory is not capable of storing the partial bitstream at once, since that would undermine our initial assumption that only the *DynMem* has enough space to store the partial bitstream. Note that, in practice, if the *DynPart* is large enough, which is the case in our proof-of-concept implementation, there are not enough BRAMs in the FPGA to store the entire partial bitstream.

After the *DynPart* is completely (re)configured, the *Prv* computes the MAC of the entire configuration memory. Therefore, the ICAP reads out the memory frame per frame, in an order chosen by the *Vrf*. For each frame, a new step in the MAC calculation is computed. Before the first step, the MAC is initialized. When the entire configuration is read out and included in the MAC computation, the MAC is finalized. The *Prv* sends back the checksum. The *Vrf* then compares the received value to a locally generated golden reference.

In practice, there is a complication that needs to be overcome to implement the above procedure. The bitstream that is sent to the FPGA does not exactly correspond to the data that the ICAP reads from the configuration memory. The reason is that the ICAP also reads out the content of all registers, which depends on the current state of the running FPGA application. Since the scope of this work is the attestation of the FPGA configuration, the *Vrf* needs to be able to make a comparison of the checksum generated by the *Prv* with the locally generated golden reference, and therefore, the register content needs to be masked out. When creating bitstreams using the Xilinx tools, this mask, which we call Msk, can be generated. We apply the Msk on the side of the *Vrf*. Therefore, the *Prv* does not only send back the MAC value to the *Vrf*, but also the content, i.e., the frames. This way, the *Vrf* can apply the Msk to the received frames in order to compare with the golden reference. Note that another option is to send the Msk to the *Prv* whenever a frame readback is requested, such that the Msk can be applied to the configuration memory content before each MAC step. This would lead to a similar communication latency: the frames would not need to be sent from *Prv* to *Vrf*, but the Msk values for each frame would need to be sent from *Vrf* to *Prv*.

In more detail, the attestation of the FPGA configuration occurs by a repetition of three commands that are sent by the *Vrf* to the *Prv*:

1) ICAP_config(*frame*): update the configuration memory with the *frame* data, which contains both the configuration memory address and the content that needs to be written;
2) ICAP_readback(*frame_nb*): read out the content of the configuration memory at the address given by *frame_nb*, send back the content to the *Vrf* and compute the next step in the MAC calculation (in case this is the first step in the MAC calculation, it is preceded by the MAC initialization);
3) MAC_checksum: finalize the MAC computation and send back the checksum to the *Vrf*.

The low-level communication steps are shown in Figure 9. The attestation protocol is initiated by the *Vrf*, who sends ICAP_config commands to the *Prv*. First, the *Vrf* instructs the ICAP to configure the intended application in the *DynPart* by transmitting the corresponding frames (from frame_m to frame_n). The number of frames that is sent this way depends on the size of the *DynPart*. The second step in the dynamic configuration is the update of the nonce, which consists of 64 bits in our implementation.

After these initial steps, the entire *DynMem* is (over)written. Next, the *Vrf* sends the ICAP_readback command to the *Prv* together with a frame address, telling the ICAP to read out a frame from the configuration memory and to perform a calculation step in the computation of the MAC. Before the first calculation step, an initialization of the MAC computation is done. The frame addresses are applied starting from address i, where i is chosen by the *Vrf*, up to address 28,487, and then from address 0 up to address i-1. The *Vrf* chooses the starting address i. In Figure 9, %28,488 is used to indicate a modular reduction with modulus 28,488. This way, all the frames in the configuration memory are included in the computation of the MAC. It is pointed out that this ascending order, starting from an offset i, is in no way required. The order in which the frames are read back can be any permutation. If desirable by the *Vrf*, a number of frames could also appear multiple times. Note that the use of a nonce already guarantees the freshness of the MAC computation, so changing the order in which the frames are configured is not necessary for freshness.

When the *Vrf* sends the MAC_checksum command to the *Prv*, the MAC is finalized and the cryptographic checksum is sent to the *Vrf*. Upon verification of the checksum, the *Vrf* is assured that the configuration originates from the *Prv* and is not tampered with. Next, the *Vrf* applies the Msk to all of the received frames, thus obtaining B_{Prv}. Similarly, the *Vrf* applies the Msk to the golden reference to obtain B_{Vrf}. When the comparison of B_{Prv} and B_{Vrf} results in equality, the *Vrf* has attested the *Prv*.

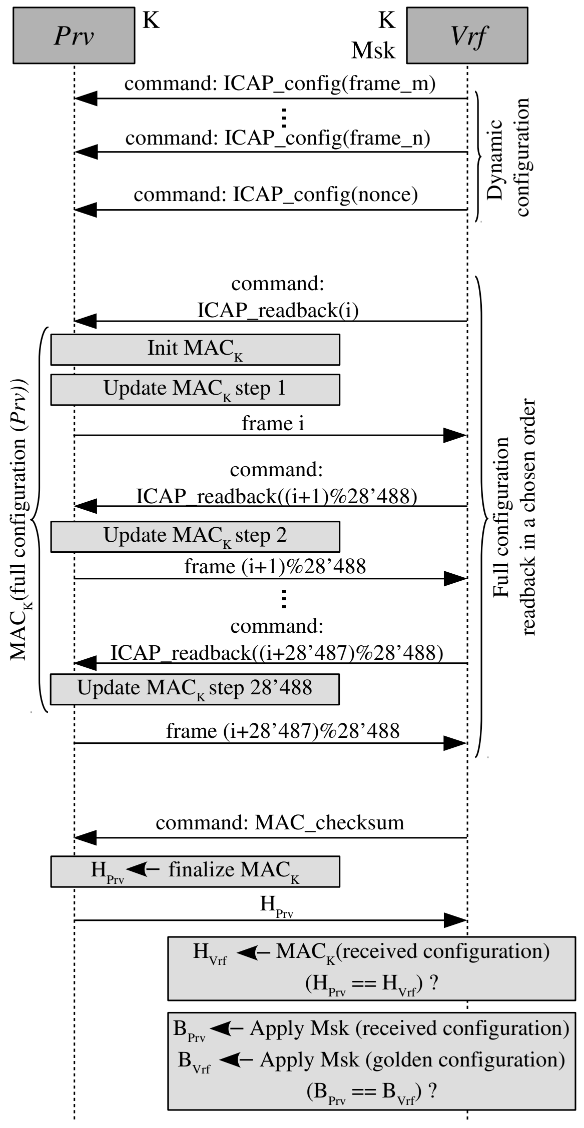

Prv
K
K
Msk
Vrf
command: ICAP_config(frame_m)
command: ICAP_config(frame_n)
command: ICAP_config(nonce)
Dynamic configuration
command: ICAP_readback(i)
Init MAC_K
Update MAC_K step 1
frame i
command: ICAP_readback((i+1)%28'488)
Update MAC_K step 2
frame (i+1)%28'488
command: ICAP_readback((i+28'487)%28'488)
Update MAC_K step 28'488
frame (i+28'487)%28'488
MAC_K(full configuration (Prv))
Full configuration readback in a chosen order
command: MAC_checksum
H_Prv ← finalize MAC_K
H_Prv
H_Vrf ← MAC_K(received configuration)
(H_Prv == H_Vrf) ?
B_Prv ← Apply Msk (received configuration)
B_Vrf ← Apply Msk (golden configuration)
(B_Prv == B_Vrf) ?

Fig. 9: Low-level communication steps, using a key K and with Msk being the mask on the communicated bitstream.

6.2 Implementation of the Architecture

The high-level view of the SACHa architecture is given in Figure 7. A block diagram of the proof-of-concept implementation of the *StatPart* is shown in Figure 10. The *StatPart* is divided into three parts that each operate in a different clock domain:

- the RX clock domain for receiving data from the *Vrf*: the RX clock is derived from the incoming network packets; it runs at 125 MHz and drives the receiving port of the ETH core and the other components in the RX clock domain;
- the ICAP clock domain for reading and writing data from/to the configuration memory: the ICAP clock is generated by the DCM; it runs at 100 MHz and drives the ICAP and the other components in the ICAP clock domain.
- the TX clock domain for transmitting data to the *Vrf*: the TX clock is generated by the Digital Clock Manager (DCM); it runs at 125 MHz and drives the transmitting port of the ETH core and the other components in the TX clock domain;

The DCM derives the TX clock and the ICAP clock from the on-board 200 MHz system clock. Note that the RX and TX clocks run at the same frequency. They cannot originate from the same clock source, though, since there might be a phase shift between the incoming and outgoing network packets. The role of the components in the three domains is explained below. The clouds between two components in Figure 10 symbolize glue logic that translates the signals coming from one component to the format expected by the other component. The ETH core provides a Gigabit network connection by receiving/transmitting one byte per cycle of the 125 MHz clock.

In the RX clock domain, the incoming network packets from the *Vrf* are received by the ETH core. The network packets are stored in the BRAM-based memory; the packets contain one of the three commands explained in Section 6.1. The Finite State Machine of the RX clock domain (RX FSM) either triggers the glue logic in the TX clock domain to initiate the running of the ICAP program or triggers the Finite State Machine of the TX clock domain (TX FSM) to transmit a network packet back to the *Vrf*.

In the ICAP clock domain, the command stored in the BRAM-based memory is executed by the ICAP. In case the stored command is ICAP_config, the ICAP takes the configuration frame, that is also stored in the BRAM, and writes it to the configuration memory. In case the stored command is ICAP_readback, the frames read out by the ICAP are stored in a FIFO, that can be read out in the TX clock domain.

In the TX clock domain, the outgoing network packets are generated. First, the packet header is loaded into a FIFO. Then, either a frame is loaded into the FIFO (by copying the content from the preceding FIFO) or the checksum generated by the MAC block (through the AES-CMAC algorithm) is loaded into the FIFO. The content of the FIFO is transmitted to the *Vrf* by the ETH core. We use 128-bit AES for the AES-CMAC algorithm, such that we need to generate a 128-bit key. In the proof-of-concept implementation, we use a key register in the *StatPart* to store the key. For a foolproof solution, a key-generating PUF needs to be implemented, as shown in Figure 7, instead of a key register.

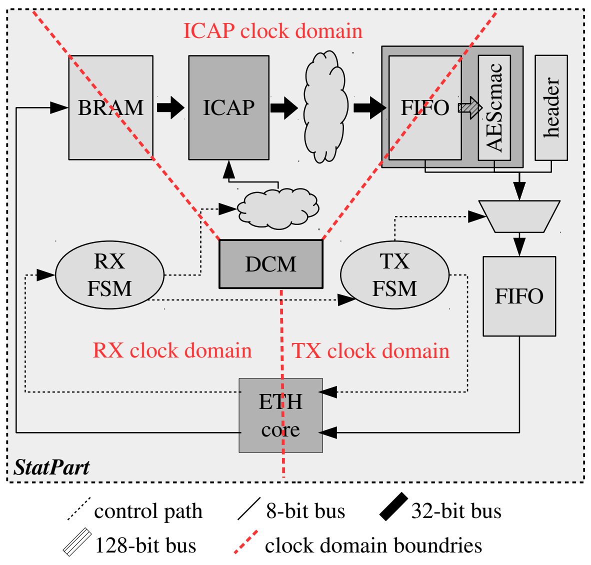


Fig. 10: The FPGA block diagram of the proof-of-concept implementation of SACHa.

7 SACHa Evaluation

7.1 Performance Evaluation

The occupied FPGA resources of the proof-of-concept implementation of the SACHa architecture on a Xilinx XC6VLX240T FPGA are presented in Table 2. The table shows the number of CLBs, (18-kbit) BRAMs, ICAPs and DCM in the considered FPGA. Further, it summarizes the occupied resources of the implemented components. The *StatPart* occupies less than 9% of the FPGA (when considering both CLBs and BRAMs). This overhead is very reasonable, since modern FPGAs are typically much larger than the FPGA used for this proof of concept.The AES-CMAC core in the *StatPart* is optimized towards low area, resulting in an implementation using 283 CLBs and 8 BRAMs (including the FIFO from which the incoming data are read). This leaves the majority of the configurable fabric to the intended application (including the nonce) in the *DynPart*.

TABLE 2: FPGA resources of the SACHa architecture.

Component	CLB	BRAM	ICAP	DCM
Entire FPGA	18 840	832	1	12
StatPart	1 400	72	1	1
MAC (+ FIFO)	283	8	0	0
DynPart	17 440	760	0	11

Table 3 shows the duration of the low-level actions in the SACHa protocol. Table 4 lists the number of times each action needs to be executed. The actions related to the configuration of a frame in the *DynMem* are repeated 26,400 times, which corresponds to the number of frames in the *DynMem*. The actions related to the readback of a frame are repeated 28,488 times, which corresponds to the total number of frames in FPGA. The initialization and finalization of the MAC need to be performed only once. The same holds for the *Vrf*'s request to compute the final checksum and the transmission of the MAC by the *Prv*. The sum of the durations of these actions is around 1.5 s. We also measured the actual duration of the execution of the SACHa protocol in a lab network, resulting in a duration of 28.5 s. From this result, we can conclude that the measured duration is dominated by the delay of the network communication. The reason for the large difference in theoretical and measured duration comes from the fact that the protocol consists of many steps (as shown in Figure 9). As a reference for the reader, it is pointed out that a direct configuration of the targeted FPGA takes around 28 s over a JTAG cable, which shows that the measured duration of our protocol is very reasonable.

TABLE 3: Timing of the low-level steps in the SACHa protocol in the proof-of-concept implementation.

Action		Time
A1	*Vrf* sends ICAP_config	8 856 ns
A2	*Prv* performs ICAP_config	1 834 ns
A3	*Vrf* sends ICAP_readback	13 616 ns
A4	*Prv* performs ICAP_readback	24 044 ns
A5	*Prv* performs MAC init	120 ns
A6	*Prv* performs MAC update	128 ns
A7	*Prv* performs MAC finalize	136 ns
A8	*Prv* performs frame sendback	2 928 ns
A9	*Vrf* sends MAC_checksum	344 ns
A10	*Prv* performs MAC sendback	472 ns

TABLE 4: Total timing of the SACHa protocol in the proof-of-concept implementation.

Action	Number of times	Time
A1	26 400	0.234 s
A2	26 400	0.050 s
A3	28 488	0.388 s
A4	28 488	0.685 s
A5	1	0.120 µs
A6	28 488	3.646 ms
A7	1	0.136 µs
A8	28 488	0.083 s
A9	1	0.344 µs
A10	1	0.464 µs
Theoretical duration		1.443 s
Measured duration		28.5 s

7.2 Security Evaluation

We use the classification of adversaries introduced in Section 3 to evaluate the security of our SACHa proposal. We consider the following threats:

- A local adversary, e.g., the owner of the FPGA platform, adds a malicious hardware module to the *DynPart* of the *Prv*'s FPGA: since the configuration data sent by the *Vrf* can only be stored in the configuration memory, the malicious hardware module has to be overwritten, which is then proven to the *Vrf*, making the attack infeasible.
- A local adversary adds a malicious hardware module to the *StatPart* of the *Prv*'s FPGA: the *StatPart* is made as small as possible, containing only relevant components for the communication and the calculation of the MAC. Since the size of the *StatPart* cannot be changed after deployment, it is impossible for the *Adv* to use the *StatPart* for communication and MAC computation while at the same time running additional malicious blocks.
- A local adversary impersonates the *Prv*: the key is only contained in the legitimate device (*Prv*) and never exchanged over a public channel, such that the MAC cannot be computed on another device (*Prv*). The fact that a PUF is used to generate the key for the MAC prevents the *Adv* from impersonating the *Prv*.
- A local adversary connects another computing device to the *Prv*'s FPGA, such that the MAC can be computed on that device and the FPGA can run malicious code: the bitstream reflects which FPGA pins are connected to peripherals, such that the *Vrf* exactly knows if there are additional connections to external devices.
- A local adversary performs a replay attack: the presence of the nonce in the initial dynamic configuration challenge makes the replay attack detectable by the *Vrf*. Further, the order in which the *Vrf* triggers the readback of the configuration frames determines the order of the steps in the MAC computation, which changes the MAC in each repetition of the protocol, even if the *Adv* manages to prevent the nonce from being updated.

8 Conclusion and Future Work

This paper proposes an architecture and protocol for the Self-Attestation of Configurable Hardware (SACHa). The mechanism allows the use of FPGAs as trusted hardware modules in hardware-based attestation schemes. Whereas these schemes usually rely on trusted tamper-resistant dedicated hardware modules, FPGAs are configurable after deployment thus inherently not tamper-resistant. Therefore, the main contribution of the paper is that it is the first work that does not assume that the FPGA is a tamper-resistant hardware module in hardware-based attestation schemes. The proposed solution consists of a novel FPGA architecture, suitable for implementation on an off-the-shelf FPGA, and attestation protocol.

Implementation results and measurements are performed on a proof-of-concept implementation on a Xilinx Virtex 6 FPGA. The experiments show that the SACHa architecture occupies less than 9% of the configurable resources on the considered FPGA. The attestation of the complete configuration memory of the FPGA based on the SACHa protocol takes 1.5 seconds (without taking into account the network delay). When the network delay is taken into account, it takes 28.5 seconds to execute the protocol in a lab setup.

The next step will be to also take the content of the registers of the running application into account (which is filtered out in the current solution by the use of a mask). This makes it possible to not only attest the FPGA configuration, but also the current state of the FPGA application. Consequently, the trend of embedding softcore processors in an FPGA can be followed, allowing the attestation scheme to do a combined verification of the FPGA configuration and the current state of the FPGA application (including the state of the embedded processor).

Another possible extension is to add a signature mechanism to the system when it is not possible to exchange a secret key between the prover and the verifier before deployment.

References

[1] D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure," in *ESORICS'10*. Springer, 2010, pp. 643–662.

[2] J. Vliegen, M. Rabbani, M. Conti, and N. Mentens, "Sacha: Self-attestation of configurable hardware," in *to appear in DATE'19*. IEEE, 2019.

[3] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "Invited: Things, trouble, trust: on building trust in IoT systems," in *DAC'16*, 2016, p. 121.

[4] Wired, "Stuxnet Incident," https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet, 2010.

[5] D. Spinellis, "Reflection as a mechanism for software integrity verification," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 51–62, 2000.

[6] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE S&P '04*, 2004, pp. 272–282.

[7] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, "Remote software-based attestation for wireless sensors," in *ESAS'05*. Springer-Verlag, 2005, pp. 27–41.

[8] Y.-G. Choi, J. Kang, and D. Nyang, "Proactive code verification protocol in wireless sensor network," in *ICCSA'07*. Springer-Verlag, 2007, pp. 1085–1096.

[9] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the Integrity of PERipherals' Firmware," in *ACM CCS'11*, 2011.

[10] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." in *NDSS '12*, 2012, pp. 1–15.

[11] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys '14*, 2014, p. 10.

[12] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny Trust Anchor for Tiny Devices," in *DAC '15*, 2015, pp. 1–6.

[13] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "DARPA: Device attestation resilient to physical attacks," in *ACM WiSec'16*, 2016, pp. 171–182.

[14] F. Kohnhäuser, N. Büscher, S. Gabmeyer, and S. Katzenbeisser, "SCAPI: a scalable attestation protocol to detect software and physical attacks," in *ACM WiSec'17*, 2017, pp. 75–86.

[15] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "SeED: Secure Non-Interactive Attestation for Embedded Devices," in *ACM WiSec'17*, 2017, pp. 64–74.

[16] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *IEEE S&P '97*, 1997.

[17] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *USENIX Security Symposium*, 2004.

[18] P. England, B. Lampson, J. Manferdelli, and B. Willman, "A trusted open platform," *Computer*, vol. 36, no. 7, pp. 55–62, 2003.

[19] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," *2009 IEEE/IFIP DSN'09*, pp. 115–124, 2009.

[20] S. Drimer and M. G. Kuhn, "A protocol for secure remote updates of FPGA configurations," in *Reconfigurable Computing: Architectures, Tools and Applications*, J. Becker, R. Woods, P. Athanas, and F. Morgan, Eds. Springer Berlin Heidelberg, 2009, pp. 50–61.

[21] A. Braeken, J. Genoe, S. Kubera, N. Mentens, A. Touhafi, I. Verbauwhede, Y. Verbelen, J. Vliegen, and K. Wouters, "Secure remote reconfiguration of an FPGA-based embedded system," in *ReCoSoC'11*. IEEE, 2011, pp. 1–6.

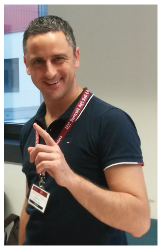
[22] J. Vliegen, N. Mentens, and I. Verbauwhede, "Secure, remote, dynamic reconfiguration of FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 4, p. 35, 2015.

[23] R. Chaves, G. Kuzmanov, and L. Sousa, "On-the-fly attestation of reconfigurable hardware," in *FPL'08*, 2008, pp. 71–76.

[24] J. Vliegen, N. Mentens, and I. Verbauwhede, "A single-chip solution for the secure remote configuration of FPGAs using bitstream compression," in *ReConFig'13*. IEEE, 2013, pp. 1–6.

Jo Vliegen received the masters degree in engineering technology from Catholic University College Limburg, Diepenbeek, Belgium, in 2005 and the PhD degree in engineering technology from the KU Leuven, Leuven, Belgium, in 2014. After three years in industry, he returned to the university college in 2008 and started his research on the reconfigurability of FPGAs. Since 2014, he has been a postdoctoral researcher with the COSIC Research Group, KU Leuven. His main research activities focus both on the implementation of cryptographic primitives on FPGAs, and on the use of (finegrained) reconfigurability of FPGAs.

Md Masoom Rabbani is a PhD student under the supervision of Prof. Mauro Conti in SPRITZ research group at University of Padova, Italy. He is affiliated in the Brain, Mind Computer Science (BMCS) school of University of Padova. After his master's degree, he worked in IBM India Pvt Ltd as an Application Developer from 2013 to 2016. In 2016 he joined the University of Padova as a PhD student. His research interests predominantly include security privacy and more precisely Remote attestations and its various techniques. Currently, he is working with Prof. Nele Mentens at KU Leuven.



Mauro Conti is Full Professor at the University of Padua, Italy, and Affiliate Professor at the University of Washington, Seattle, USA. He obtained his Ph.D. from Sapienza University of Rome, Italy, in 2009. After his Ph.D., he was a Post-Doc Researcher at Vrije Universiteit Amsterdam, The Netherlands. In 2011 he joined as Assistant Professor the University of Padua, where he became Associate Professor in 2015, and Full Professor in 2018. He has been Visiting Researcher at GMU (2008, 2016), UCLA (2010), UCI (2012, 2013, 2014, 2017), TU Darmstadt (2013), UF (2015), and FIU (2015, 2016, 2018). He has been awarded with a Marie Curie Fellowship (2012) by the European Commission, and with a Fellowship by the German DAAD (2013). His research is also funded by companies, including Cisco, Intel and Huawei. His main research interest is in the area of security and privacy. In this area, he published more than 250 papers in topmost international peer-reviewed journals and conferences. He is Area Editor-in-Chief for IEEE Communications Surveys Tutorials, and Associate Editor for several journals, including IEEE Communications Surveys Tutorials, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, and IEEE Transactions on Network and Service Management. He was Program Chair for TRUST 2015, ICISS 2016, WiSec 2017, and General Chair for SecureComm 2012 and ACM SACMAT 2013. He is Senior Member of the IEEE.

Nele Mentens received her master and Ph.D. degree from KU Leuven in 2003 and 2007, respectively. Currently, Nele is an associate professor at KU Leuven in the COSIC group at the Electrical Engineering Department (ESAT). Her research interests are in the domains of configurable computing for security, design automation for cryptographic hardware and security in constrained environments. Nele was a visiting researcher for 3 months at the Ruhr University Bochum in 2013 and at EPFL in 2017. She was/is the PI in around 15 finished and ongoing research projects with national and international funding. Nele was general co-chair of FPL'17 and program chair of PROOFS'18 and EWME'18. She served in around 50 program committees and was an associate editor for the IET Information Security journal. Nele is (co-)author in around 100 publications in international journals, conferences and books. She is a senior member of the IEEE.