SACHa: Self-Attestation of Configurable Hardware

Jo Vliegen[1], Md Masoom Rabbani[2], Mauro Conti[2] and Nele Mentens[1]
[1]ES&S and imec-COSIC/ESAT, KU Leuven, Belgium, Email: firstname.lastname@kuleuven.be
[2]SPRITZ, University of Padua, Italy, Email: lastname@math.unipd.it

***Abstract*— Device attestation is a procedure to verify whether an embedded device is running the intended application code. This way, protection against both physical attacks and remote attacks on the embedded software is aimed for. With the wide adoption of Field-Programmable Gate Arrays or FPGAs, hardware also became configurable, and hence susceptible to attacks (just like software). In addition, an upcoming trend for hardware-based attestation is the use of configurable FPGA hardware. Therefore, in order to attest a whole system that makes use of FPGAs, the status of both the software and the hardware needs to be verified, without the availability of a tamper-resistant hardware module.**

In this paper, we propose a solution in which a prover core on the FPGA performs an attestation of the entire FPGA, including a self-attestation. This way, the FPGA can be used as a tamper-resistant hardware module to perform hardware-based attestation of a processor, resulting in a protection of the entire hardware/software system against malicious code updates.

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are silicon chips that can be configured after fabrication. They provide embedded systems with a higher performance per consumed energy unit than general-purpose microprocessors, while offering configurability after deployment. 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. The high-volume production of FPGAs makes it possible for vendors to closely follow the latest technology at a reasonable cost. Furthermore, the continuous efforts in the past decades to integrate application-specific building blocks, close the performance gap with ASICs. Thanks to these evolutions, the use of FPGAs in embedded systems is growing.

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 IV. 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. The mechanism

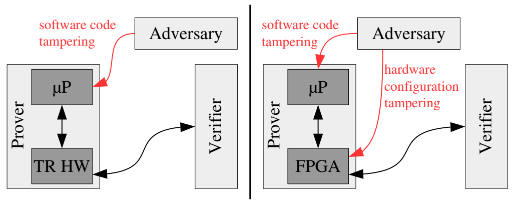

Fig. 1. Adversary models in traditional hardware-based attestation (left) and in this paper (right), where µP and TR HW denote the microprocessor and the tamper-resistant hardware module, respectively.

that is proposed, is inspired by the work of Perito and Tsudik [8], 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 configuration memory, it is far from straightforward to apply the results of [8] directly to FPGAs. Our solution, which we call SACHa (Self-Attestation of Configurable Hardware), consists of (1) a novel hardware design, mapped on an off-the-shelf FPGA, and (2) a communication protocol executing the attestation process based on the proposed FPGA design.

II. Preliminaries

A. FPGA Features used in SACHa

An FPGA can be logically partitioned, and a specific partition can be configured separately while other partitions continue to operate normally. The part of the configuration memory that configures a specific partition is then updated at run-time. 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 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. Next to the static partition, there can be one or more run-time configurable partitions, which are referred to as dynamic partitions. The ICAP can write a bitstream into the parts of the configuration memory that are connected to the dynamic partitions, resulting in the reconfiguration of these partitions. Such a bitstream, only targeting a dynamic partition, is referred to as a partial bitstream.

Since we target the detection of malicious faults in the configuration memory, SACHa also uses the configuration memory readback mechanism, which allows the ICAP to read out the entire configuration memory.

B. Attestation Concept of SACHa

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.

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 (bounded/limited) memory. The original proposal, as introduced by Perito and Tsudik in [8], 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. 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. To do so, we overcome the challenges that occur due to the differences between embedded processors and FPGAs. The resulting architecture uses partial reconfiguration and configuration memory readback to ensure 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.

III. System and Adversary Model

Fig. 2 shows the entities in the attestation scheme and the components of the system on the prover's side.

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 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 [8], 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.

Prv, Vrf	prover and verifier, respectively
Adv	adversary
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

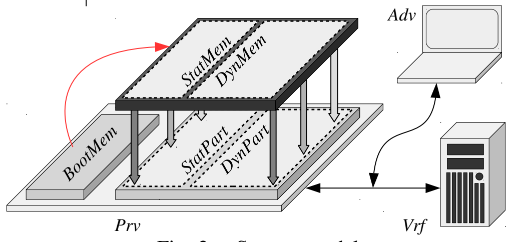

Fig. 2. System model.

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. We exclude Hardware Trojan insertion from the attack space. We also consider side-channel analysis attacks and physical attacks that actively modify the configurable fabric or the FPGA-based system out of our current scope.

IV. Related Work

We explore related work in remote attestation in this section. The discussed methods mainly belong to either software (SW) based or hardware (HW) based attestation.

In general, most of the SW-based mechanisms do not require HW support and rely on a strict time-bound challenge-response protocol, e.g., [10]. Other software-based schemes like [12], [11], [3], [7] rely on empty memory filling or sequential memory readout. SW-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.

HW-based methods predominantly rely on the use of specialized HW. Schemes like [9], [5], [6] employ a Trusted Platform Module (TPM) to safeguard source code execution details. Although HW-based attestation provides better security than SW-based schemes, the associated cost makes them unsuitable for low-end embedded devices.

We are aware of two papers that deal with the remote attestation of configurable hardware [4], [2]. Both papers rely on an external memory that is accessible during the attestation process and/or a tamper-proof configuration memory. Further, our work is also related to secure remote configuration of FPGAs, as described in [4], [1], [14].

Compared to the work in [4], [2], 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.

V. Our Solution: SACHa

A. Contribution

The mechanism we introduce in this paper improves the security of FPGA-based attestation. 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 and attestation protocol.

B. FPGA Architecture

We apply the bounded memory model, introduced in [8] and summarized in Section II-B, 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 [13], 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, this automatically implies that the configurable fabric of the FPGA is running the application intended by the *Vrf*. The platform used in [8] 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 3 gives a high-level overview of the FPGA architecture.

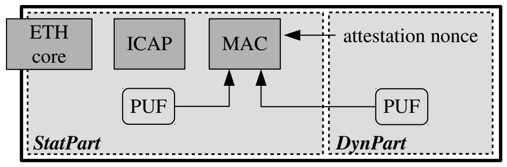

Fig. 3. High-level FPGA architecture of SACHa, in which the key for the MAC comes either from *StatPart* or from *DynPart*.

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. 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; (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 weak PUF in our solution; attacks/weaknesses of PUFs are considered out of scope here. The use of a PUF leads to an additional enrollment step in the preparation of the FPGA. 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 device is 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, 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.

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 V-B.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.

C. Attestation Protocol

Figure 4 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 V-B.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 4. 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.

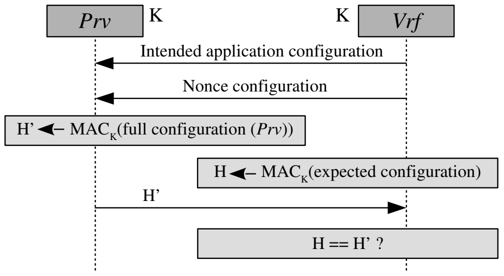
Prv
K
K
Vrf
Intended application configuration
Nonce configuration
H' ← MAC_K(full configuration (Prv))
H ← MAC_K(expected configuration)
H'
H == H' ?

Fig. 4. SACHa protocol, using a key K.

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

VI. 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.

A. Implementation of the Protocol

The configuration memory of the FPGA consists of 28,488 frames, where one frame contains 81 words of 32 bits. Since we want to make the *StatPart* as small as possible, we use a BRAM 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 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.

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 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*.

B. Implementation of the Architecture

The high-level view of the SACHa architecture is given in Figure 3. A block diagram of the proof-of-concept implementation of the *StatPart* is shown in Figure 5. The *StatPart* is divided into three parts that each operate in a different clock domain: (1) 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; (2) the *ICAP clock domain* for reading and writing data from/to the configuration memory: the ICAP clock is generated by the DCM (Digital Clock Manager); it runs at 100 MHz and drives the ICAP and the other components in the ICAP clock domain; (3) the *TX clock domain* for transmitting data to the *Vrf*: the TX clock is generated by the 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. The RX and TX clocks run at the same frequency, but cannot originate from the same clock source, 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 5 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 VI-A. 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. If 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. If 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 3, instead of a key register.

VII. SACHa Evaluation

A. 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 I. The *StatPart* occupies less than 9% of the FPGA (when considering both CLBs and BRAMs). This overhead is very reasonable, since

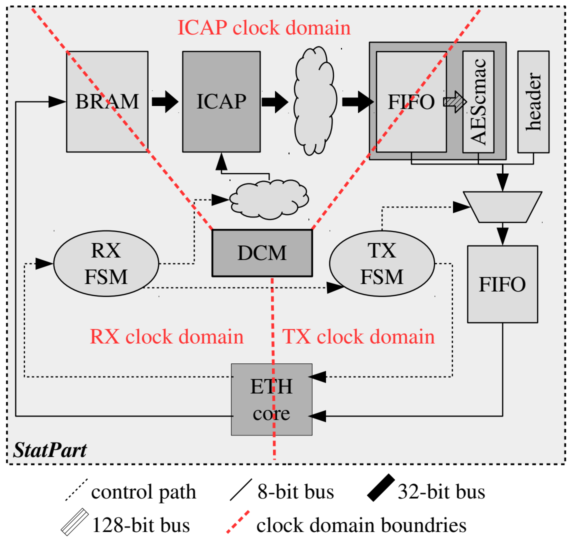

Fig. 5. FPGA architecture of the SACHa proof-of-concept implementation.

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 I
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 II shows the duration of the low-level actions in the protocol and lists the number of times each action is executed. The actions related to the configuration of a frame in the *DynMem* are repeated 26,400 times, corresponding to the number of frames in the *DynMem*. The actions related to the readback of a frame are repeated 28,488 times, corresponding to the total number of FPGA frames. 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 to transmit the MAC. The total duration 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 a separate communication step for each bitstream frame. As a reference for the reader, it is pointed out that a basic remote 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 II

TIMING OF THE LOW-LEVEL STEPS IN THE THE PROOF OF CONCEPT.

	Action	Time	# times	Duration
A1	*Vrf* sends ICAP config	8.9 μs	26 400	0.23 s
A2	*Prv* does ICAP config	1.8 μs	26 400	0.05 s
A3	*Vrf* asks for ICAP readback	13.6 μs	28 488	0.39 s
A4	*Prv* does ICAP readback	24.0 μs	28 488	0.69 s
A5	*Prv* initializes MAC	0.1 μs	1	0.1 µs
A6	*Prv* updates MAC	0.1 μs	28 488	3.6 ms
A7	*Prv* finalizes MAC	0.1 μs	1	0.1 µs
A8	*Prv* sends back frame	2.9 μs	28 488	0.08 s
A9	*Vrf* sends MAC checksum	0.3 μs	1	0.34 µs
A10	*Prv* sends back MAC	0.5 μs	1	0.46 µs
			Theoretical duration	1.44 s
			Measured duration	28.5 s

B. Security Evaluation

We consider the following threats:

- A malicious hardware module is added 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 module has to be overwritten, which is then proven to the *Vrf*, making the attack infeasible.
- A malicious hardware module is added to the *StatPart* of the *Prv*'s FPGA: the *StatPart* is made as small as possible, containing only relevant components, so it is impossible to change it in a way that allows the *Adv* to compute the correct MAC and at the same time run malicious code on the FPGA.
- The *Adv* 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*.
- Another computing device is connected 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.
- The *Adv* performs a replay attack: the use of a nonce makes the replay attack detectable by the *Vrf*. Further, the order in which the *Vrf* reads back the configuration frames determines the order of the steps in the MAC computation, which guarantees a fresh MAC even if the *Adv* manages to prevent the nonce from being updated.

VIII. CONCLUSION AND FUTURE WORK

The SACHa mechanism proposes a solution for the self-attestation of an FPGA. This way, the FPGA can be used as a trusted hardware module in hardware-based attestation schemes. These schemes usually rely on the presence of a trusted tamper-resistant dedicated hardware module. SACHa allows FPGAs, which are configurable after deployment and thus inherently not tamper-resistant, to be used inside these trusted hardware modules. The 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.

We have proven the efficiency and effectiveness of our approach based on a proof-of-concept implementation on a Virtex 6 FPGA. The SACHa architecture occupies less than 9% of the configurable resources on the considered FPGA. The execution of the SACHa protocol to attest the complete configuration memory of the FPGA (without taking into account the network delay) takes 1.5 seconds. The duration of the protocol measured in a lab network is 28.5 seconds.

The next step will be to also attest the state of the FPGA application. This way, the trend of embedding softcore processors in an FPGA can be followed, allowing the attestation scheme to verify both the FPGA configuration and the state of the embedded processor at once.

Further, it is assumed that a key is shared between the prover and the verifier before deployment, which obviates the need for the online authentication of the entities. In some application scenarios, however, it could be interested to add, e.g., a signature mechanism to the system.

REFERENCES

[1] 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*, pages 1–6. IEEE, 2011.

[2] R. Chaves, G. Kuzmanov, and L. Sousa. On-the-fly attestation of reconfigurable hardware. In *FPL'08*, pages 71–76, 2008.

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

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

[5] P. England, B. Lampson, J. Manferdelli, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[6] 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*, pages 115–124, 2009.

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

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

[9] 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.

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

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

[12] D. Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62, 2000.

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

[14] J. Vliegen, N. Mentens, and I. Verbauwhede. Secure, remote, dynamic reconfiguration of FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(4):35, 2015.