Remote attestation of bare-metal microprocessor software: a formally verified security monitor

Jonathan Certes[0000−0001−8773−3749] and Benoît Morgan[0000−0003−3223−3995]

IRIT-ENSEEIHT, University of Toulouse - France
firstname.lastname@irit.fr

Abstract. Remote attestation is a protocol to verify that a remote algorithm satisfies security properties, allowing to establish dynamic root of trust. Modern architectures for remote attestation combine signature or MAC primitives with hardware monitors to enforce secret confidentiality. Our works are based on a verified hardware/software co-design for remote attestation, VRASED. Its proof is established using formal methods and its implementation is conducted on a simple embedded device based on a single core microcontroller. A heavy modification of the core, along with a hardware monitor, enforces security properties.
We propose to extend this method to microprocessors where cores cannot be modified. In this paper, we tackle this problem with support from the microprocessor's debug interface and demonstrate that the same security properties also hold.

Keywords: Remote attestation · Security · Formal verification · FPGA

1 Introduction

Remote attestation consists in verifying that a machine called *Prover* satisfies necessary security properties to be trusted by a remote machine called *Verifier* [4]. These security properties are generally verified through a challenge-response protocol as depicted on figure 1.

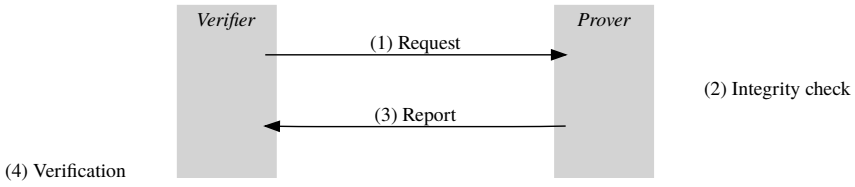

Fig. 1. Remote attestation protocol

At first, the *Verifier* transmits a challenge to the *Prover* to request the attestation (1). The *Prover* computes an authenticated integrity check over its

memory and the challenge (2) and reports the result to the *Verifier* (3). From this result, the *Verifier* checks whether the *Prover* memory state is valid (4).

To authenticate the answer, remote attestation requires the execution of a cryptographic challenge in a corrupted environment. Safety for this execution is mandatory to ensure security for the protocol. Authentication for the answer follows one of the following strategies:

1. either the challenge is designed so that a modification of its execution environment from the adversary inevitably alters the computation of the answer (which affects computation time) ;
2. or an integrity-dependent transformation for the *Prover* is kept secret from the adversary.

We can cite Checkmate [8] and Pioneer [13] as representative of the works from the first category. Works such as SMART [7] and VRASED [12] clearly fit in the second category as they imply maintaining a secret and require hardware support for access control.

Formal methods bring a high level of trust in verified remote attestation security. They allow to establish a proof, based on axioms or demonstrated properties, that the system and its implementation are secure. Formal verification is generally conducted in three steps. First, the system (hardware or software) is modelled, for example as an automaton [14]. Then, properties to be satisfied by the system are formally described, this includes secret confidentiality for remote attestation. In the end, *model-checking* conducts an exhaustive state exploration approach or a proof demonstrates that the system verifies the properties.

Section 2 summarizes the state of the art about remote attestation, usually implemented on simple devices, and its formal verification. An approach to extend verified remote attestation to microprocessors is descried in section 3, the contribution is summarized in section 4 and a verified security monitor is detailed in section 5.

2 State of the art

Eldefrawy et al. proposed SMART [7], a hardware modification for the *Prover* on microcontroller Texas Instrument MSP430. They dedicated a protected memory region to store the secret. Lugou et al. [10] tried to propose a unified method to verify hardware/software co-designs. They applied this method on SMART and modelled the system with *Proverif*. This method does not scale and comes with imprecisions as explained by Eldefrawy et al. in [6]. In all these works, security properties are enforced by a hardware extension, monitoring the system and capable of restarting it in case of a leak of the secret in its next state. *Model-checking* formally verifies the safety for this monitor.

De Oliveira Nunes et al. proposed VRASED (*Verifiable Remote Attestation for Simple Embedded Devices*) [12], a hardware/software co-design method for remote attestation. Their approach includes the implementation of an attesting

software and a hardware monitor. This union guaranties security properties. The implementation is also conducted on microcontroller MSP430.

The attesting software is based on a formally verified cryptographic library (memory safety, functional correctness and secret independence) [15], which computes the *HMAC* of an attested region from a shared secret. The hardware monitor enforces access control to the secret as well as immutability and atomicity for the attesting software. A heavy modification of the core enables the use of the interruption signals, program counter and read/write addresses as inputs for the hardware monitor. The model for the attacker is as follows: the attacker can control the entire software state of the *Prover*, code and data, that is not explicitly protected by the hardware monitor.

The following steps describe how soundness and security proofs are obtained:

1. Soundness and security are expressed using temporal logics. This includes:
   - any direct access to the secret can only be performed by the attesting software (access control) ;
   - any memory region written by the attesting software (excluding the final results of the computation) cannot be read by the attacker.
2. Properties are described in the same formalism as soundness and security. This includes:
   - The resulting *HMAC* computed by the attesting software is correct (functional correctness).
   - The attesting software cannot be modified by the attacker (immutability).
   - Execution of the attesting software cannot be interrupted, it starts at the first instruction and finishes at the last (atomicity).
3. Proof for soundness and security are obtained through rewritings with *Spot* [5]: when their implication by the conjunction of the properties is a tautology.
4. *Model-checking* with *NuSMV* [3] ensures that the hardware monitor verifies the properties. Conversion tool *Verilog2SMV* [9] translates *Verilog* Hardware Description Language (HDL) into an automaton which is checked against the properties.
5. Some properties are axiomatic as they are converted from the specifications of the formally verified cryptographic library.

This co-design method for remote attestation is adapted to simple embedded devices such as microcontrollers. Our goal is to re-use its proofs for soundness and security as a framework to secure remote attestation on microprocessors.

3 Extension to microprocessors

Unless working with open-source architectures, it is impossible to conduct modifications on the hardware, thus microprocessor cores. As a consequence, equivalent observations must be achieved from other inputs to deduce the state of the interruption signals, program counter and read/write addresses.

3.1 Environment

Modern Systems on Chip (SoC), such as Xilinx Zynq-7000, integrate ARM microprocessors along with programmable logic in a single device. This combines the flexibility and the parallelism of a Field-Programmable Gate Array (FPGA) with the performances of an Application-Specific Integrated Circuit (ASIC). Spatial partitioning for sensitive memory (such as the secret or the code of the attesting software) and implementation of a hardware extension can take place in the FPGA.

ARM microprocessors come with a debug interface called *CoreSight* which enables real-time instruction flow tracing without slowing down execution. Traces contain information to reconstruct the execution of a program. During the execution of the attesting software, the activation of program flow tracing, combined with the addition of specific instructions, provides data that can be used for monitoring. In particular, *Program Trace Macrocell* (PTM) module outputs a trace when an interruption or an indirect branch occurs and provides the destination address [2].

In order to enforce access control and immutability properties, the hardware extension in the FPGA monitors access signals to the sensitive memory. To enforce atomicity properties, it is coupled with the use of the debug interface. A trace is generated when an interruption occurs during the execution of the attesting software. The first and last instructions are chosen such as a trace is generated and gives the value of the program counter: an indirect branch with the address of the next instruction as a destination.

3.2 Refinement approach

Extension to microprocessors comes with new constraints and increases the capabilities, thus power of the attacker. To adapt, we follow a stepwise refinement approach, in which we start from a very abstract microprocessor, similar to microcontrollers, down to a model close from reality.

First, we consider single-core microprocessors where we abstract capabilities for the attacker to rely on cache, Memory Management Unit (MMU) and the configuration of *CoreSight*. This system is called model 0: functionalities and targeted applications are identical to those of a microcontroller. The definitions of the soundness and security are identical to the ones described in [12]. They are to be proven.

Then, in a new refined model, a new constraint is added to the definition of security ; for example: "the attacker is capable to reconfigure *CoreSight*". The conjunction of a new property, described in temporal logic, is added to the specification. As a consequence, the system must satisfy this new property so that the proof stays valid. Two possible approaches are considered:

- Either the new system is translated in an automaton and the verification of (both old and new) properties are conducted through *model-checking*. This approach is identical to the one introduced by [12], it is adapted to small automata.

– Or a simulation relation is established: if all states of the old model are simulated in the new one, then the new model verifies the same properties [11]. Only the new properties are to be verified through *model-checking*. This approach reduces the effort of verification in case of space-state explosion.

These operations are repeated for each new capability provided by the use of a microprocessor. At each iteration, proof is established on the new model. The refinement of the specification can be verified at each step with an implication of its previous expression.

4 Contribution

To re-use proofs from VRASED and extend them to microprocessors, we re-use their attesting software and automaton as-is. Due to the immutability of the core, the inputs of this automaton: interruption signals, program counter and read/write addresses, have to be deduced from external observations.

In this paper, we propose a solution based on hardware/software co-design to deduce the value of these useful signals at critical steps of the protocol execution. Axioms describe the behaviour of *CoreSight* in accordance with its documentation and the content of the software. New properties are verified through *model-checking* of the hardware. Then, we conduct a proof to show that the soundness and security properties from [12] also hold on our model 0, even with no modification of the core.

5 Verified security monitor

In this section we describe a method to prove the soundness and security on model 0. Targeted applications are identical to microcontrollers, i.e. embedded systems, and no hardware modification for the core is required.

5.1 Attacker model in practice

An application is loaded in the DDR, it runs bare-metal on one core of the system and behaves as a code loader. It expects code coming from the Universal Asynchronous Receiver-Transmitter (UART), to load it in the memory and execute it. Depending on the application, this software might also expect code to be transferred from the network interface. Any code can be loaded so that the attacker can control the entire software state of the *Prover*. Memory is limited so that an entire operating system does not fit.

5.2 Attesting software and monitor architectures

Both the secret and the code of attesting software are stored in dedicated ROM located in the FPGA. A RAM is also located in the FPGA and is used as an exclusive stack by the attesting software. *CoreSight* debug interface outputs

program flow traces trough a trace port interface unit to the FPGA. Signals are monitored by the hardware extension to obtain read/write addresses and information about the execution of attesting software.

Figure 2 represents this implementation of a Xilinx Zynq-7000 where partitioned memories are accessible through the Advanced eXtensible Interface (AXI) communication interface [1].

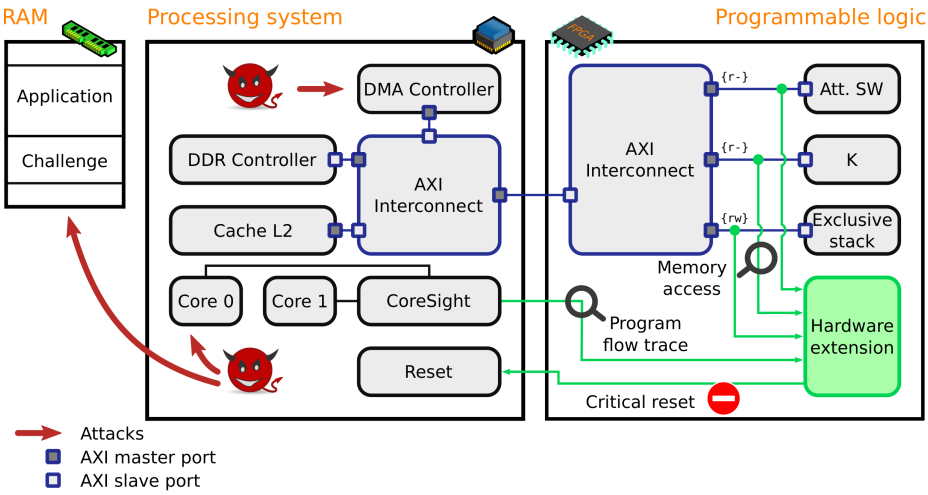

Fig. 2. Implementation on Xilinx Zynq-7000

Attesting software is re-used from [12] and wrapped around with specific instructions to configure *CoreSight* and the MMU. As a consequence, during its execution:

- first instruction is an indirect branch that forces *CoreSight* to output the value of the program counter.
- MMU restricts read/write addresses to the memories located in the FPGA and the challenge location.
- an interruption outputs a trace with exception information.
- last instruction is an indirect branch that forces *CoreSight* to output the value of the program counter.

5.3 Definition of model 0

To validate our approach, a proof is conducted on model 0, where some of the capabilities of the attacker are abstracted. Here are hypothesis that we assume in the definition of our model:

1. the whole system is synchronous.

2. when a trace containing a destination address is output by *CoreSight*, the program counter takes this address at the next clock cycle; that is, the next system state.
3. the attacker does not reprogram the FPGA.
4. the attacker does not reconfigure *CoreSight* or the MMU.
5. during the execution of the attesting software by one core of the microprocessor, other cores are paused.
6. cache and registers are empty before and after an execution of the attesting software and after a reset.

These hypotheses are expressed using temporal logics and are considered axiomatic. They help proving that security properties hold and will be discarded in the subsequent models. Future refined versions of the hardware extension must verify that they are still enforced.

5.4 Proof strategy

The hardware extension is described in *Verilog* and converted into an automaton. It contains the hardware monitor from [12]. Since the processor is left unmodified, verified properties from [12] are expressed using deduced values for the monitored signals (not their real values) and cannot be used to prove the remote attestation security anymore. We add axioms and prove that their conjunction implies the initial VRASED security properties. For instance, an axiom can be "if the deduced value of the program counter is in the address range of the attesting software, then its real value is". These axioms form a proof obligation to be discharged by the other modules of the system.

The other modules of the system aim at obtaining *CoreSight* traces and access signals to memories in the FPGA, then process them to deduce values that are accepted by VRASED original automaton. The cornerstone of the deduction process is a transducer that translates the content of *CoreSight* traces into deduced values for the interruption signals, program counter and read/write addresses. The automaton depicted on figure 3 represents parts of this transducer. Its outputs are predicates where:

- pc_d represents the deduced value for the program counter
- irq_d represents the deduced value for the interruption signal
- CR (for *Critical Region*) is the address range where the attesting software is located
- CR_{min} is the address of the first instruction of the attesting software
- CR_{max} is the address of the last instruction of the attesting software

Its inputs are events. For each label on the transitions, the event is that *CoreSight* outputs a compressed trace containing branch information. Predicates for these labels are defined as follows:

- $@CR_{min}$: destination address is equal to CR_{min}
- $@CR_{max}$: destination address is equal to CR_{max}

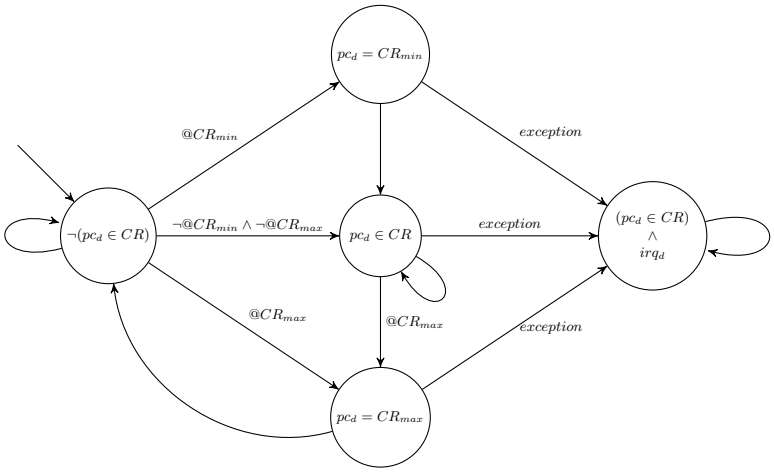

Fig. 3. Transducer *CoreSight* to VRASED : deduction of *pc*

– *exception* : trace contains exception information

This transducer verifies properties in which deduced values are expressed using the decoded address and exception information. To demonstrate our proof obligation, new axioms must then be added to express this decoded information according to the real values of the program counter and the interruption signals. With the conjunction of these new axioms and the verified properties, we prove the implication of the proof obligation.

These axioms form the next proof obligation for the rest of the system. The hardware is extended to decode addresses and exception information from a decompressed trace delivered by *CoreSight*. This leads to new properties, verified with *model-checking*, that helps to prove the next proof obligation with other axioms.

As a consequence, the design of the hardware extension is an iterative process which can be described as follows:

1. Security properties are defined in [12]. We re-use the same automaton to verify a property P_n expressed from deduced predicates.
2. Axioms A_n are necessary to imply the security property. These axioms form a proof obligation.
3. Axioms A_n are implied by the conjunction of properties P_{n+1} and axioms A_{n+1}. Rewritings with a theorem prover demonstrate that this implication is a tautology.
4. The hardware monitor is extended to verify properties P_{n+1}. This hypothesis is ensured with *model-checking*.
5. Steps 3 and 4 are repeated while incrementing n until the proof is based only on axioms resulting from a translation of *CoreSight* documentation.

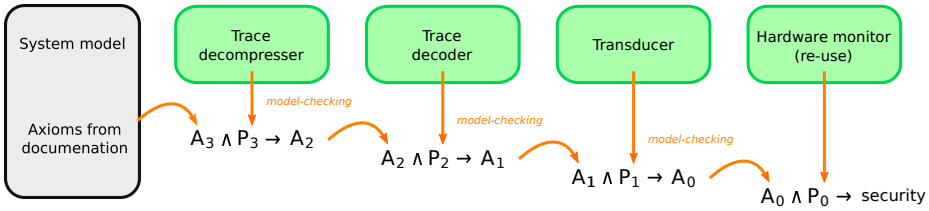

Fig. 4. Proof obligations

At the end of the design process, our hardware extension is a composition of hardware modules: a re-used automaton from [12], a transducer, a trace decoder and a trace decompresser. Figure 4 summarizes our proof strategy.

The role of the trace decompresser is to identify the type of packet that is transmitted by *CoreSight* and inform the transducer that an event occurred. The trace decoder transmits decoded addresses and exception information to the transducer when an event occurs.

5.5 Results and future work

Once we re-used the automaton from [12], our hardware extension allows to deduce all predicates and prove that all security properties from [12] also hold on our model 0. As a consequence, remote attestation security for embedded systems applications has been proven. Soundness has also been formally verified on our model 0 following the same approach as in [12].

To extend formally verified remote attestation to microprocessors, our objective is to refine our model so that each hypothesis defined in section 5.3 is discarded and the attacker capabilities are not restricted. We intend to follow an iterative approach and extend our system to verify new security properties as described in section 3.2.

We also aim at targeting complex applications such as software running on an operating system. The method for remote attestation may be adapted to the application in order to stay sound and secure.

6 Conclusion

VRASED proposed a formally verified hardware/software co-design method, based on *model-checking* and proof, and is implemented on a simple architecture with a heavy modification of the core.

Our approach to extend this method to ARM microprocessors is as follows: *CoreSight* debug interface proceeds to a non-invasive collect of data that traces the execution flow. Combined with a hardware extension and software wrapping, we retrieve appropriate inputs to re-use VRASED hardware monitor. Regarding hypothesis on the attacker capabilities, we have proven that soundness and security for the remote attestation can be ensured for embedded systems applications with no modification of the CPU core.

To extend the use of microprocessors to remote attestation, a refinement approach allows to achieve a formal verification in an iterative manner. When *model-checking* leads to a space-state explosion, we establish a simulation relation between automata and prove that the system meets its specification. Soon, our approach will allow to prove remote attestation soundness and security for embedded systems applications with non-restricted capabilities for the attacker.

References

1. AMBA AXI and ACE Protocol Specification, no. IHI 0022D ID102711 (2003-2011)
2. CoreSight PTM-A9 Technical Reference Manual - Revision: r1p0, no. ARM DDI 0401C ID073011 (2008-2011)
3. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv version 2: An opensource tool for symbolic model checking. In: Proc. International Conference on Computer-Aided Verification (CAV 2002). vol. 2404. Springer, Copenhagen, Denmark (July 2002)
4. Coker, G., Guttman, J.D., Loscocco, P., Herzog, A.L., Millen, J.K., O'Hanlon, B., Ramsdell, J.D., Segall, A., Sheehy, J., Sniffen, B.T.: Principles of remote attestation. Int. J. Inf. Sec. **10**(2) (2011)
5. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. Lecture Notes in Computer Science, vol. 9938. Springer (Oct 2016)
6. Eldefrawy, K., Nunes, I.O., Rattanavipanon, N., Steiner, M., Tsudik, G.: Formally verified hardware/software co-design for remote attestation. arXiv preprint arXiv:1811.00175 (2018)
7. Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D.: Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In: NDSS. vol. 12 (2012)
8. Ghosh, A., Sapello, A., Poylisher, A., Chiang, C.J., Kubota, A., Matsunaka, T.: On the feasibility of deploying software attestation in cloud environments. In: 2014 IEEE 7th International Conference on Cloud Computing (2014)
9. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2smv: A tool for word-level verification. In: 2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016 (2016)
10. Lugou, F., Apvrille, L., Francillon, A.: Toward a methodology for unified verification of hardware/software co-designs. Journal of Cryptographic Engineering (2016)
11. Milner, R.: An algebraic definition of simulation between programs. In: Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971 (1971)
12. Nunes, I.D.O., Eldefrawy, K., Rattanavipanon, N., Steiner, M., Tsudik, G.: VRASED: A verified hardware/software co-design for remote attestation. In: 28th USENIX Security Symposium. USENIX Association, Santa Clara, CA (Aug 2019)
13. Seshadri, A., Luk, M., Perrig, A., van Doom, L., Khosla, P.K.: Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In: Malware Detection (2007)
14. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings) (1995)
15. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: Hacl*: A verified modern cryptographic library. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS, Dallas, TX, USA (2017)