Formal Verification of Embedded Systems for Remote Attestation

G. Cabodi, P. Camurati, C. Loiacono, G. Pipitone, F. Savarese and D. Vendraminetto
Politecnico di Torino
Dipartimento di Automatica e Informatica
Turin, Italy

Abstract: Embedded systems are increasingly pervasive, interdependent and in many cases critical to our every day life and safety. As such devices are more and more subject to attacks, new protection mechanisms are needed to provide the required resilience and dependency at low cost. Remote attestation (RA) is a software-hardware mechanism that securely checks the internal state of remote embedded devices. This protocol is executed by: (1) a *prover* that, given a secret key and its actual state, generates a result through an attestation algorithm; (2) a *verifier* that, given the key, the expected prover actual state, accepts or rejects the result through a verification algorithm. As the security of a protocol is only as good as its weakest link, a comprehensive validation of its security requirements is paramount. In this paper, we present a methodology for formal verification of hardware security requirements of RA architectures. First we perform an analysis and a comparison of three selected RA architectures, then we define security properties for RA systems and we verify them using a complete framework for formal verification.

Key–Words: Formal Verification, Security, Remote Attestation, Embedded Systems

1 Introduction

Remote attestation (RA) is the process of securely verifying the internal state of a remote hardware platform. It can be performed either statically (at boot time) or dynamically, at run-time in order to establish a dynamic root of trust[1].

In this paper we focus on three RA architectures SMART [1] , Sancus [2] and TrustLite [3], as they are interesting w.r.t. formal verification aspects.

SMART is a simple approach, based on hardware-software co-design, for establishing a dynamic root of trust in a remote embedded device. SMART focuses on low-end micro-controller units (MCU) that lack specialized memory management or protection features. It requires minimal changes to existing MCUs, while providing concrete security guarantees, and assumes few restrictions on adversarial capabilities [1].

Sancus can remotely attest to a software provider that a specific software module is running undamaged, and it can authenticate messages from software modules to software providers. Software modules can securely maintain local state, and can securely interact with other software modules that they choose to trust. Sancus achieves these security guarantees without trusting any infrastructural software on the device. The Trusted Computing Base (TCB) on the device is only the hardware. Moreover, the hardware cost of Sancus is low [2].

TrustLite presents mechanisms for secure exception handling and communication between protected modules, enabling seamless interoperability with untrusted operating systems and tasks. TrustLite scales from providing a simple protected firmware runtime to advanced functionality such as attestation and trusted execution of userspace tasks. In contrast with the previous approaches, it also solves the problem of handling memory access violations and hardware interrupts. Like Sancus, TrustLite is an FPGA prototype showing that the above capabilities are achievable even on low-cost embedded systems [3].

Following the work presented in [4] we propose a methodology for formal verification of hardware security requirements in RA architectures. The key insight is that many security requirements can be formulated as taint-propagation properties, since this is the most natural way of expressing properties related to information flow and access control. A taint-propagation property has the following elements:

- *source (src)* : RTL signals “seeded” with the taint
- *destination (dest)* : signals not to be reached by the taint in order to satisfy the security requirements
- *conditions*: temporal logic expressions that must be true at various points in the taint propagation, e.g.,

[1]The term dynamic root of trust refers to approaches for providing evidence for a trustworthy platform state, i.e., root of trust, at more arbitrary points in time than just start-up, i.e., dynamic.

when the taint starts or when it ends

Using this notation, confidentiality requirements can be verified by setting a hardware secret as the src and the data bus of an external interface as the dest. Similarly, integrity can be verified by setting an untrusted interface as the src and a sensitive signal as the dest.

In this work, we perform formal analysis of taint-propagation properties using PdTrav [5] [6] and VIS (Verification Interacting with Synthesis) [7] formal verification tools. The formal verification tool we used analyzes taint-propagation properties and either proves each property or finds a counterexample showing a functional path from src to dst. To sum up we translate several high-level security requirements into taint-propagation properties and prove them using PdTrav and VIS.

Our approach entails the following major contributions:

- performing a thorough analysis and a comparison of the RA architectures mentioned previously w.r.t. security and formal verification aspects
- defining security properties that represent the core security requirements for RA architectures
- presenting a complete framework for performing verification of formal security properties related to RA architectures
- showing the verification results of security properties on one of the selected RA architectures, comparing different model checkers and verification techniques. Whenever a security property verification fails, we present the corresponding attack that could affect the security of the system

The rest of the paper is organized as follows. Section 2 presents background notions. Section 3 describe the guidelines to define taint-properties for RA architectures. Section 4 presents a complete formal verification framework, in order to verify previously defined taint-properties. Section 5 illustrates experimental results. Finally, Section 6 concludes the paper with some summarizing remarks.

2 Background

2.1 RA Notations

Since our interest is to apply formal methods to verify security properties of Remote Attestation architectures, in this section we describe all the background information of these different areas.

Definition 1 - Remote attestation protocol: we use the term Remote Attestation to denote a protocol, whereby a *Verifier* verifies the internal state of another device called a *Prover*. This protocol is executed over a network. The goal of the protocol is to allow a not damaged *Prover* to create an authentication token, that convinces the *Verifier* that the former is in some well-defined (expected) state. Whereas, if the *Prover* has been damaged and its state has been modified, the authentication token must reflect this. The generic RA protocol is presented in Figure 1.

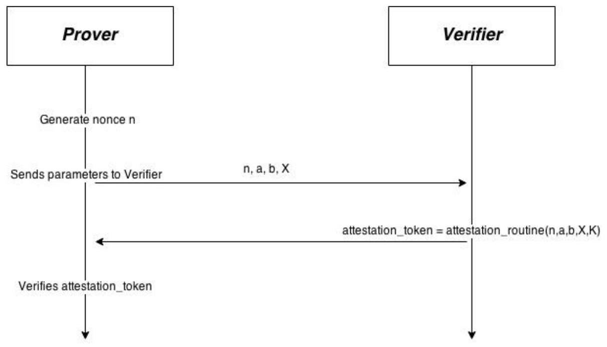
Prover
Verifier
Generate nonce n
Sends parameters to Verifier
n, a, b, X
attestation_token = attestation_routine(n,a,b,X,K)
Verifies attestation_token

Figure 1: Remote attestation protocol

Definition 2 - Infrastructure Provider (IP): IP, owns and administers a (potentially large) set of microprocessor-based systems. A variety of third-party Software Providers SPs are interested in using the infrastructure provided by IP. They deploy Software Modules SMs on the nodes administered by IP.

Definition 3 - Trusted Computing Base (TCB): The TCB of a task is the set of components (hardware and software) that must be secure to assure the unmodified execution of that task.

Definition 4 - Trustlet: trusted tasks which are designed to accomplish a particular security mechanism.

2.2 RA architectures

In the following we highlight the peculiarities of the RA architectures under analysis.

SMART: In the SMART architecture the attestation algorithm uses the hashing algorithm HMAC, which is located in a Trusted Region *TR* in ROM; whereas the secret key *k* is stored in a particular RAM location. To enable secure Remote Attestation SMART implements two control accesses on the memory bus controller: 1) *k* can be accessed only by the instructions located in the TR 2) an instruction can accesses *TR* only from the initial TR address and it can leaves *TR* only from its last address.

As a result, the secret key is only accessible by the trusted code stored in ROM and (except for first instruction in *TR*) if the actual Program Counter points into the *TR* region, then the previous Program Counters value must point also into *TR*.

Sancus: This RA architecture consists of an Infrastructure Provider *IP* that manages a set of microprocessor-based nodes N and a Software Provider SP_i that deploys Software Modules *SM* into N: these software modules contain a text section (protected code and constants) and protected data section. *IP* generates (exploiting the key generation functions) and it manages three types of secret keys: 1) K_N, shared between node *N* and *IP* 2) $K_{N,SP}$, shared between a provider *SP* and node *N* 3) $K_{N,SP,SM}$, shared between a node *N* and a provider *SP*, only accessible by *SM*. A software provider *SP* can employ Remote Attestation to verify that the correct software module *SM* is running on an expected node *N* through the module key $K_{N,SP,SM}$. The attestation code is saved into the text section of *SM* and the memory access control logic (bus controller) ensures that: 1) The protected data section of module is only accessible while code in the text section of that module is begin executed 2) The code in the text section can only be executed by jumping to a well-defined entry point.

TrustLite: the TrustLite platform consists of a SoC that includes a set of trustlets with their critical data regions and a Memory Protection Unit (MPU) that ensures access controls on all memory acesses: memory is organized into a number of protected regions with associated access permissions, kept in local registers available to the MPU. Trustelets can use Remote Attestation to inspect and validate the local platform states.

These selected RA architectures will be analyzed and used in the following sections as case studies for formal security properties verification.

2.3 Model checking notation

Considering the formal verification aspects, we address systems modeled by labeled state transition structures and represented implicitly by Boolean formulas. From our standpoint, a system M is a triplet $M = (S, S_0, T)$, where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is a total transition relation. The system state space is encoded with an indexed set of Boolean variables $X = \{x_1, \ldots, x_n\}$, so that a state $s \in S$ corresponds to a valuation of the variables in X, and a set of states can be represented with a Boolean formula over X. A literal is a Boolean variable or its negation. A clause is a disjunction of literals. A CNF formula is a conjunction of clauses. Most modern SAT solvers [8, 9] adopt clauses as their main representation and manipulation formalism for Boolean functions. Whenever necessary, given a Boolean function F, we will use notation F_{CNF} for its CNF representation.

Given a sequential system M and an invariant property p, SAT-based Bounded Model Checking (BMC) [10] is an iterative process to check the validity of p up to a given bound. To perform this task, the system transition relation T is unrolled k times

$$T^k(X^{0..k}) \quad = \quad \bigwedge_{i=0}^{k-1} T(X^i, X^{i+1})$$

to implicitly represent all state paths of length k. After that, BMC tools may implement variants of SAT checks, such as:

$$bmc^k(X^{0..k}) \quad = \quad S_0(X^0) \wedge T^k(X^{0..k}) \wedge \bigvee_{i=0}^{k} \neg p(X^i)$$

The check looks for counterexamples (of length $\leq k$) falsifying p, starting from set of the initial states S_0.

Although BMC tools are effective at finding bugs, even in large designs, their verification method is not complete, since it only guarantees the correctness of a property up to the given bound. Therefore, specific techniques are required in order to support Unbounded Model Checking (UMC). The ability to check reachability fix-points is thus the main difference (and additional complication) between BMC and UMC. All UMC approaches basically rely on one or more methods able to detect that the forward, backward or mixed reachability analysis and/or circuit unrolling they perform is complete.

3 Defining Security properties for RA architectures

We analyzed the architecture and design of a generic RA hardware and selected three broad areas to verify, we converged on a set of properties which represent the core security requirements for each area. In Section 5 we report security properties for the selected RA architectures and present the verification results.

Key Secrecy: Key is the encryption or description key, used in RA application flows. Prover and verifier can use Key for encryption or decryption by configuring a crypto-engine to get the key directly from HW. However, to reduce the attack surface, the user is not allowed to read the Key.

In the SMART architecture the secret key k is stored in a particular memory location in RAM and it is only accessible by the attestation code, saved in a trusted region *TR* in ROM. All controls relative to key secrecy are implemented in BUS CONTROLLER and concern the Program Counter PC. More in details, when some operation wants to read *k*, BUS CONTROLLER monitors PC checking if it belongs to trusted region *TR*; as the result, *k* can be accessed only from within ROM-resident SMART code. But this control is not enough to guarantee key secrecy. In fact, during SMART code running, some intermediate results are saved in the CPU registers: it is necessary to prevent key leakage during attestation code execution; an attacker can stop SMART during its running and retrieve essential information about *k* through bits inside registers. Therefore another control is added inside BUS CONTROLLER: SMART code cannot be interrupted before its end and it can be only invoked at its beginning.

In the Sancus architecture three types of keys K_N,$K_{N,SP}$, $K_{N,SP,SM}$ are managed. All keys are produced by the infrastructure provider *IP*, using key generation functions *kdf*s, that, using a pseudo-random function, derive secret keys from a master key or in general, from a secret value such as a password or a pass-phrase; this secret value can be used in conjunction with parameters that are not secret or common to a group of users. *Kdf*s are broadly utilized because their use can avoid that an attacker, who owns the derived key, learns considerable information about the secret value in input. To guarantee key secrecy it is possible to find different strategies that consider the generation of some node master keys. 1) In a first option a single node master key is employed for all nodes of the system that are protected through the same key; this strategy is very simple but not very reliable, because an endangerment of the node master key can alter the security of entire system 2) A second option includes the presence of a node master key also, but, in this case, it is derived from a master key of the *IP*, through a key derivation function that uses the identifier of the node (when a node and a software provider are registered in the infrastructure, they receive unique public identifiers *N* and *SP* respectively) 3) A third option is the most common one among Sancus architectures. It consists in generating different random keys for each node. These keys are securely saved in *IP*. As a result, the master node key K_N is only managed by *IP* and the hardware. When a software provider is registered, it receives its key $K_{N,SP}$ = *kdf* (K_N,*SP)* from IP which can then be used by the node *N* to create $K_{N,SP,SM}$ = *kdf*($K_{N,SP}$,*SM)* that is specific to the module SM loaded on *N* by *SP*. Note that: 1) K_N is only known by *IP*; 2) $K_{N,SP}$ is known by *IP* and *SP*; 3) $K_{N,SP,SM}$ is known by *IP*, *N* (each node keeps a protected storage area for all these keys) and it can be computed by *SP* since it received $K_{N,SP}$ from and since it knows the identity *SM* of the software module.

The TrustLite architecture performs Remote Attestation using trustlets. Trustlets ensure that the OS or other software cannot manipulate the outcome of memory read accesses of the MPU register set or other trustlets code regions. Then, at least two kinds of keys are managed: 1) K_t, shared between two trustlets. It is used by a trustlet to inspect and validate the status of other trustlets on the common platform and to establish a mutually authenticated and confidential communication channel. 2) K_s, shared between a trustlet and MPU, used to validate the memory state of the local MPU. The structures of the MPU and of the trustlets guarantee simply that the access to their keys can be performed exclusively by themselves.

Mode Separation: In RA architectures the execution of some routine, e.g. the attestation code, should be done only in supervisor mode. Some operations and resources in the SMART architecture are accessible by the supervisor only, to guarantee key isolation, memory safety and atomic execution of ROM code. The main goal of controls inside the BUS CONTROLLER is to avoid key leakage; in other words, in user mode it is never possible to read the secret key stored in RAM: the access to this address is allowed to SMART code only. Any user's software has the total control of RAM memory, except for key location, after and before SMART code, can modify any writable code, learn any secret that is not explicitly protected by BUS CONTROLLER and it can invoke SMART whenever it wants, but it cannot interrupt ROM code, which must be executed atomically and cannot be invoked partially. Therefor the passage from user mode to supervisor mode occurs when a user's application calls SMART code in ROM and finishes with the end of itself.

Focusing on the Sancus architecture, it is clear that a distinction between user and supervisor mode is necessary within the software modules *SM* inside the nodes *N*. The attestation code, as it has already been stated, is saved inside software modules (text section) and it can be used by a software provider to verify that a *SM* is the one expected on node *N* through the secret key $K_{N,SP,SM}$ (kept into data section). In this way, any user cannot access to the protected data of software module; this area is accessible during supervisor mode only, specifically while the attestation code, in the text section, is running. Moreover each node securely saves all keys $K_{N,SP,SM}$ in a protected storage area, which can be read and modified exclusively by the super-user. Therefore any user can manipulate all software on the nodes, except the protected storage area; he can also run the attestation code, but only by jumping to a well-defined entry point, then moving to supervisor mode.

In the TrustLite architecture the MPU performs the important task controlling user accesses to system memory and detecting protection violation from them. This is possible thanks to a write-protected table called *Trustlet Table* that records data regions with associated access permissions. Thus, the MPU can be programmed by the OS for the next respective task to be scheduled. Obviously, the controls (in the MPU and trustlets), that use this table, are focused to avoid that, during user mode, it is possible to read or modify the secret keys necessary to attestation and to avoid violation caused by not-permitted access in some protected regions.

Avoiding Denial of Service (DoS) Attacks :
A primary concern is to avoid Denial-of-Service Attacks: in order to achieve it, sensitive transactions from untrusted Infrastructure Providers shall not be able to write to internal registers, to reset the CPU or to sweep all RAM memory. A further issue is that a transaction initiated by local user software is considered untrusted too. Considering the SMART architecture, if a verifier wants to execute a RA procedure, it sends some information to the User Application (UA) in execution on the prover, which will save these inputs in a predefined location in the RAM and will execute the SMART code (stored inside a Trusted Region TR located in the ROM), in order to output the final checksum C. Assuming that the attacker has complete control over the communication channel (between prover and verifier) and over the prover (before and after executing SMART), DoS attacks could be achieved in three different scenarios:

1. the attacker, pretending to be the verifier, sends manipulated input values, like assigning to the return address of the SMART procedure an address belonging to the attestation code (located in ROM) given that the Bus Controller monitors only the Program Counter, this attack causes an infinite loop (without any security property violation)

2. the attacker could manipulate all the input parameters stored in RAM, before the execution of TRs code, in order to invalidate the final result

3. after the execution of the SMART procedure, the attacker could manipulate the final value of C, if it is performed before UA reads it.

It is clear that these attacks exploit data sent by the verifier.

4 Verification Framework for RA architectures

In this section we present a complete model checker tool and the verification methodologies useful to verify the class of properties introduced in the previous section.
The "Politecnico di Torino" Reachability Analysis and Verification (PdTRAV) package is a set of MC engines oriented to evaluate and to benchmark new algorithmic ideas. It may represent a good starting point for experimental evaluation and comparison, as it won some of HWMCC [11] competitions. PdTRAV supports several symbolic reachability and MC methods:

- BDD-based representations and traversals, including forward, backward, combined (approximate) forward/ (exact) backward algorithms [12, 13] partitioned BDDs and/or image computation procedures [14, 15]

- Interpolant-based verification, with ad-hoc abstraction and tightening techniques [16, 17], integrated SAT-based approaches [18, 19], Interpolant reduction techniques [20] and Guided Refinement [21]

- Property Directed Reachability (PDR) verification strategies [22]

- Inductive reasoning (inductive invariants [23]) and symbolic manipulation of And-Invert Graphs (AIGs [24]), with circuit-based quantification [25]

- BMC with AIG-based circuit compaction (before moving to CNF-based SAT calls) but without incremental SAT

The tool also supports model transformations and reductions, typically activated as pre-processing steps of MC procedures.
We use PdTRAV's different verification strategies to verify the properties described in the following.

To simplify the properties presentation we first define some variables:

1. $addr_f$ and $addr_l$ represent respectivily the first and the last address of the ROM memory (sROM).

2. $is\_in\_sROM = (PC \geq addr_f \wedge PC \leq addr_l)$; PC belongs to the sROM address space

3. $was\_in\_srom = (PPC \geq addr_f \wedge PPC \leq addr_l)$; PC for the previous instruction pointed to the sROM

Key Secrecy The key is stored in a specific RAM location that is protected by SMART features. Moreover the Key location can be read by an instruction present in the sROM address space only. As described in Section 3 the key is required for the attestation and should be secret to avoid treats. The key secrecy property is guaranteed using the bus controller in two stages: (1) by monitoring the address bus, in order to find out unauthorized accesses to the key location; (2) by checking the actual program counter, in order to attest if the current instruction belongs to the sROM address space and have the rights to read the key.

To verify the above property, we formalize it in LTL logic:

$$G((is\_in\_sROM \vee \neg(address\_bus = key\_address))$$

We finally present a property to enforce the key security. We first model the crypto engine as follows:

$$att_{res} = \Omega(K, inputs_{xored}, mem_{xored})$$

then:

$$K = \overline{\Omega}(inputs_{xored}, mem_{xored}, att_{res})$$

where:

- Ω is a cryptography operator and $\overline{\Omega}$ his inverse
- $inputs_{xored}$ is the xor of all inputs provided to the SMART procedure
- mem_{xored} is the memory xor of locations specified as inputs to SMART
- att_{res} is the result of the attestation procedure stored in the RAM memory

We state that a generic user cannot obtain the key as result of the above attestation procedure. The corresponding LTL property is:

$$G(att_{exec} \wedge (RAM[res_{loc}] \neq \Omega(inputs_{xored}, mem_{xored}, K)))$$

where:

- att_{exec} boolean value indicating the execution of the attestation routine
- res_{loc} is the memory location attestation result

This property is satisfied as long as the Ω operator is a not reversible.

Mode separation In RA architectures the execution of the attestation code should be performed only in supervisor mode. This property is satisfied by the bus controller checking accesses to the sROM. The access is valid as long as the ROM is accessed at the first location. The first instruction present in ROM disables the interrupts. The context switching is allowed only if the attestation is completed.

The property is formalized in LTL logic and checks all the possible access configurations. All possible access configurations are represented as a conjunction of clauses. In the following we present the list of clauses.

Statement 1
The current program counter PC and the previous program counter PPC are both inside the RAM and not in the sROM:

$$\neg is\_in\_sROM \wedge \neg was\_in\_sROM$$

Statement 2
The current program counter PC and the previous program counter PPC are both inside the sROM and not in the RAM:

$$is\_in\_sROM \wedge was\_in\_sROM$$

Statement 3
PC is in the sROM and the PC address is the first of the sROM. In other words this means that the program is entering in the sROM:

$$is\_in\_sROM \wedge address\_Bus = addr_l$$

Statement 4
The actual instruction is outside the sROM and the previous instruction was in the sROM but at the last location:

$$\neg\, is\_in\_sROM \wedge was\_in\_sROM \wedge PPC = addr_l$$

In Section 5 we report, for some properties, the verification results and the corresponding threats in case of fails.

Avoiding DoS attacks In Section 3 we described properties that can prevent DoS attacks. Moreover we defined three kinds of properties in order to detect DoS attacks. In the following, using the notations and the methodology described in Section 3, we formalize the above properties using LTL logic:

$$\neg is\_in\_sROM \vee (X, a, b, out)_{locations} \in RAM$$

This property considers the following scenario: a malicious code tries to modify the SMART input parameters, in order to alter the state of the attestation execution. More in details, if X belongs to the ROM address space, SMART will be executed continuously with no violation exceptions. Inputs a and b and out should be valid RAM addresses.

$$\neg(PC \in RAM \wedge wr_{op} \wedge att_{exec}) \vee \neg(addr_{bus} = out)$$

The above property considers the modification of the output of the attestation result. Modifying the results the attestation procedure can be violated.

In Sancus the IP corresponds to the Verifier of the SMART architecture, whereas the SP and N correspond to the Prover. For this reason the properties defined for SMART can be applied to the Sancus architecture, taking into account that in the Sancus architecture the crypto-engine is the kdf function and we need to manage three keys instead of one.

Finally, the architecture of the Trustlite Secure Loader is similar to the SMART's one, then the previous considerations can be applied also on TrustLite architecture.

5 Experimental Results

We implemented a prototype version of the SMART RA architecture on top of a simple CPU model provided by VIS-model checker. This CPU model is inspired to MIPS CPUs. Starting from this model the instruction set has been extended and other modules like Memory (RAM and ROM), as shown in Figure 2 , interrupt controller and an interrupt generator have been designed.

The crypto-engine is simulated using XOR operations, for this reason this model is vulnerable to some very simple attacks and then represents a good benchmark for our model checkers. In order to implement SMART functionalities, a bus controller, containing all the informations to detect security violation, has been implemented. Figure 3 shows our model architecture details.

We verified security properties using PdTrav and VIS tools. All our experiments ran on a Quad-core workstation, with CPU frequency at 2.5 GHz and equipped with 16 GB of main memory. The time and memory limits were always set to 1800 seconds and 4 GB, respectively. Section 5.1 shows verification results of RA security properties and compares PdTrav with VIS model checkers, in terms of verification time. Section 5.2 compares different PdTrav verification strategies.

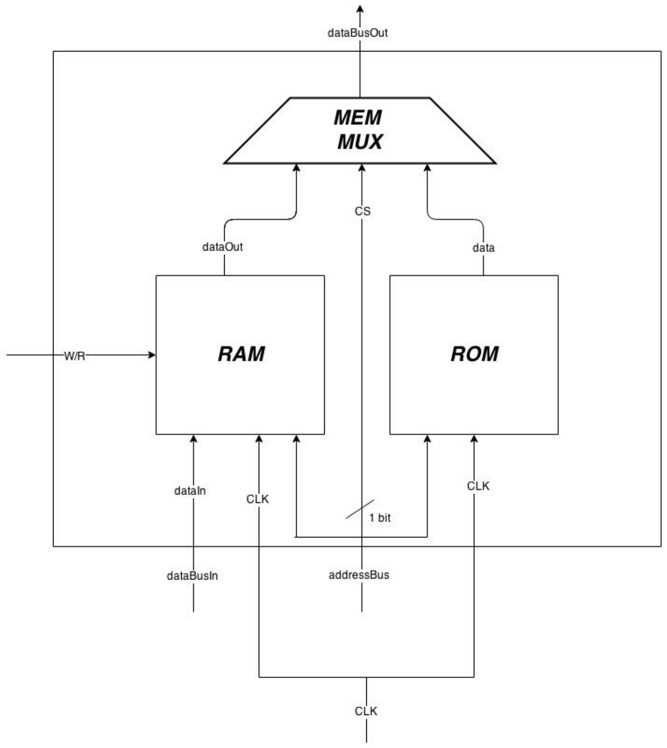

Figure 2: RA memory architecture

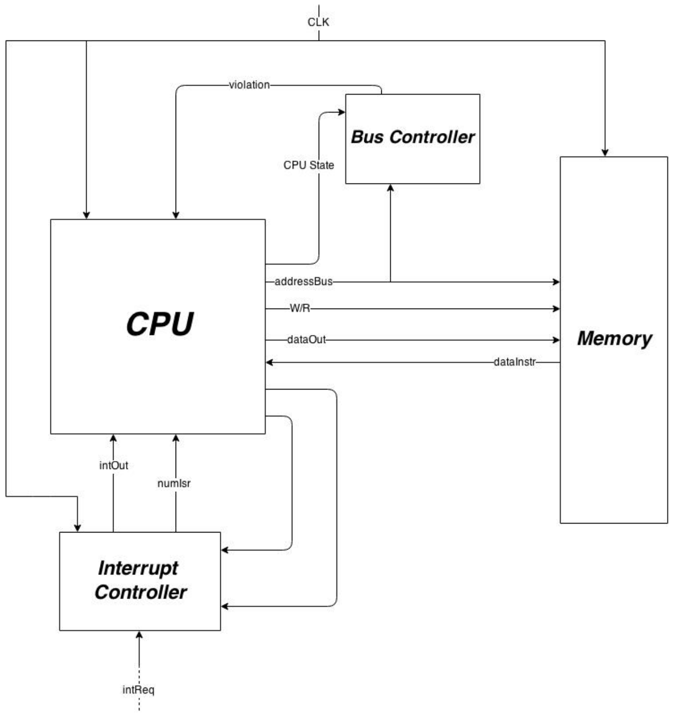

Figure 3: RA architecture model

5.1 RA Model checking

Given the security properties for the RA architectures, defined in the core sections of this paper, we present verification results for PdTrav and VIS. We verified the SMART model described previously, we also present an attack to discover the secret key in user mode.

We first perform Bounded Model Checking (BMC) setting the Bound according to properties and model characteristics.

Table 1 reports for each security property (column Property) the verification best time for VIS (column VIS_Time) and PdTrav (column PDT_Time) tools and the corresponding result (column Verif_Result). Notice that 3 properties fail and the first one is related to the key secrecy feature. In order to prove the consistency of our verification result we performed an attack related to key_prop_2. We have assumed that the adversary has complete control over the software state, code and data of the prover before and after SMART execution: the attacker can modify and know any secret that is not protected by the MCU on the prover. Among the several attacks that can be performed on the SMART architecture, we choose one in which the secret key can be obtained eluding the controls implemented in the BUS CONTROLLER. This software attack is based on two statements:

1. The memory address space of RAM and ROM is contiguous: any program in RAM can make a jump operation to the first location of the SMART code
2. The SMART code in ROM reads from RAM not only the key, but also other parameters that are not protected. Any program in RAM can modify these parameters, and it is able to perform DoS attacks and to get the key.

A critical input parameter for SMART code is *X*, that is the return address of the attestation procedure: this value is not protected by the BUS CONTROLLER, therefore it can be changed by any program in RAM; for instance, if *X* has a value belonging to address space of SMART code in ROM, an infinite loop is caused when it is running. Moreover the operations implemented by the SMART code are well known: first a memory HMAC produces the results *output test memory*, a cryptographic checksum of a region [*start address*, *end address*] in prover's memory; then this result, the key and the others input parameters saved in RAM (*A*, *B*, *X*, *n*, *out*, *x_flag*) are used to compute the final result. In particular *n* is posted by verifier to prover to avoid replay attacks, *x_flag* value determines if jump or not on *X* after attestation, *in* is an input parameter optionally used by prover, and, finally, *out* is the output address where to store the final checksum *C*, which is returned to verifier. The latter verifies correctness of *C* by recomputing it using the same parameters and the symmetrically secret key.
Algorithm 1 presents our bad code performing the attach described above. The *output test memory* value is computed using *start address* and *end address* values. Notice that computing the key is not a heavy task as the input values can be easily modified.

```
1: start_address = 0;
2: end_address = 1;
3: n = 0;
4: out = 0;
5: $x_{flag}$ = 0;
6: X = malicious code first instruction;
7: output_test_memory=executeSmart(A,B);
8: KEY=$A \oplus B \oplus output\_test\_memory$
```

Algorithm 1: Key discover attack

To sum up, the idea is to modify suitably the input parameters and to obtain *k* through final result of HMAC.

Finally we present verification results using the Unbounded Model Checking verification technique. Table 2 shows best verification time for PdTrav (column PDT_Time) and VIS (column VIS_Time). Notice that Verif_Result column reports the same results presented in Table 1. This prove that the model checker under tests provides reliable results.

5.2 Verification strategies

This section shows the results unbounded model checking runs adopting different verification strategies, implemented in PdTrav. We compare different techniques on the given set of properties and the SMART model described above.

Table 3 reports verification results considering three different unbounded model checking strategies: interpolation (ITP column), binary decision diagram reachability (BDD column) and Property Directed Reachability strategy (PDR column). Results shows that the most suitable strategies for security properties verification are ITP and PDR (see Section 4 for further details).

Property	VIS_Time [s]	PDT_Time [s]	Bound	Verif_Result
key_prop_1	12.4	**6.27**	800	PASS
key_prop_2	2.86	**1.43**	60	FAIL
mode_prop	26.54	**10.7**	800	PASS
dos_prop_1	1.3	**0.5**	20	FAIL
dos_prop_2	8.34	**4.32**	400	FAIL

Table 1: Bounded Model Checking best results for VIS and PdTrav

Property	VIS_Time [s]	PDT_Time [s]	Verif_Result
key_prop_1	66.76	**23.58**	PASS
key_prop_2	40.77	**13.43**	FAIL
mode_prop	127.34	**47.45**	PASS
dos_prop_1	40.25	**20.78**	FAIL
dos_prop_2	70.67	**31.89**	FAIL

Table 2: Unbounded Model Checking best results for VIS and PdTrav

PROP	ITP	PDR	BDD	Verif_Result
key_prop_1	27.43	**23.58**	57.34	PASS
key_prop_2	**13.43**	16.64	36.48	FAIL
mode_prop	50.63	**47.45**	105.27	PASS
dos_prop_1	25.46	**20.78**	35.32	FAIL
dos_prop_2	35.67	**31.89**	64.42	FAIL

Table 3: Unbounded Model Checking - verification PdTrav strategies results

6 Conclusions

This paper first analyzes and compares RA architectures w.r.t. security and formal verification aspects, then it presents new approaches to define and verify RA security properties. Our main contribution is to provide a methodology to verify RA architectures along with a complete framework tuned to address the model checking problem using an integrated approach, that improves over RA security verification. Overall, our framework shows good performances and flexibility, as reported in the experimental results section. Future work consists verifying complex microcontrollers and models including the Sancus and Trustlite architectures.

Acknowledgements: part of this work was performed by Francesco Savarese during an internship at Eurecom supervised by Aurelien Francillon (Networking and security dept. at Eurecom) and Ludovic Apvrille (Telecom ParisTech).

References:

[1] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik, "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," in *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, UNITED STATES, 02 2012.

[2] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C., 2013, pp. 479–498, USENIX.

[3] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *European Conference on Computer Systems (EuroSys)*. Apr. 2014, ACM.

[4] Pramod Subramanyan and Divya Arora, "Formal verification of taint-propagation security properties in a commercial soc design," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, 2014, pp. 1–2.

[5] Formal-Methods-Group, "PdTrav Tool, http://fmgroup.polito.it/index.php/download/viewcategory/3-pdtrav-package," 2014.

[6] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer, “Benchmarking a model checker for algorithmic improvements and tuning for performance,” *Formal Methods in System Design*, vol. 39, no. 2, pp. 205–227, 2011.

[7] The VIS group colorado edu, “http://vlsi.colorado.edu/ vis/index.html,” 2001.

[8] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an Efficient SAT Solver,” Las Vegas, Nevada, June 2001, IEEE Computer Society.

[9] N. Eén and N. Sörensson, “The Minisat SAT Solver, http://minisat.se,” Apr. 2009.

[10] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, “Symbolic Model Checking using SAT procedures instead of BDDs,” New Orleans, Louisiana, June 1999, pp. 317–320, IEEE Computer Society.

[11] A. Biere and T. Jussila, “The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc,” .

[12] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer, “Symbolic forward/backward traversals of large finite state machines,” *Journal of Systems Architecture*, vol. 46, no. 12, pp. 1137–1158, 2000.

[13] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer, “Mixing forward and backward traversals in guided-prioritized bdd-based verification,” in *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, 2002, pp. 471–484.

[14] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer, “Improving the efficiency of bdd-based operators by means of partitioning,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 18, no. 5, pp. 545–556, 1999.

[15] Gianpiero Cabodi, “Meta-bdds: A decomposed representation for layered symbolic manipulation of boolean functions.,” in *CAV*, Grard Berry, Hubert Comon, and Alain Finkel, Eds. 2001, vol. 2102 of *Lecture Notes in Computer Science*, pp. 118–130, Springer.

[16] Gianpiero Cabodi, Paolo Camurati, and Marco Murciano, “Automated abstraction by incremental refinement in interpolant-based model checking.,” in *ICCAD*, Sani R. Nassif and Jaijeet S. Roychowdhury, Eds. 2008, pp. 129–136, IEEE.

[17] Gianpiero Cabodi, Marco Murciano, Sergio Nocco, and Stefano Quer, “Boosting interpolation with dynamic localized abstraction and redundancy removal,” *ACM Trans. Design Autom. Electr. Syst.*, vol. 13, no. 1, 2008.

[18] Gianpiero Cabodi, Paolo Camurati, Luz Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer, “Trading-off SAT search and variable quantifications for effective unbounded model checking,” in *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, 2008, pp. 1–8.

[19] Gianpiero Cabodi, Luz Amanda Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer, “Partitioning interpolant-based verification for effective unbounded model checking,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 382–395, 2010.

[20] Gianpiero Cabodi, Carmelo Loiacono, and Danilo Vendraminetto, “Optimization techniques for craig interpolant compaction in unbounded model checking,” *Formal Methods in System Design*, vol. 46, no. 2, pp. 135–162, 2015.

[21] Gianpiero Cabodi, Marco Palena, and P. Pasini, “Interpolation with guided refinement: Revisiting incrementality in sat-based unbounded model checking,” in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, 2014, pp. 43–50.

[22] Niklas Een, Alan Mishchenko, and Robert Brayton, “Efficient implementation of property directed reachability,” in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, Austin, TX, 2011, FMCAD ’11, pp. 125–134, FMCAD Inc.

[23] G. Cabodi, S. Nocco, and S. Quer, “Strengthening model checking techniques with inductive invariants,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 1, pp. 154–158, Jan 2009.

[24] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi, “Circuit-based boolean reasoning.,” in *DAC*. 2001, pp. 232–237, ACM.

[25] Gianpiero Cabodi, Marco Crivellari, Sergio Nocco, and Stefano Quer, “Circuit based quantification: Back to state set manipulation within unbounded model checking,” in *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, 2005, pp. 688–689.