IoT Security and Privacy: Remote Attestation

Alessandro Brighente

Master Degree in Cybersecurity

UNIVERSITÀ DEGLI STUDI DI PADOVA

SPRITZ SECURITY & PRIVACY RESEARCH GROUP

The Software Problem

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- IoT devices have their own software and this can be compromised by malicious entities
- It is not easy to detect attacks in on-field devices, as Stuxnet showed
- We need solutions to attest the legitimacy of (software and hardware) components by distantly looking at them and their behavior
- We need to account for the constrained resources of these devices

Remote Attestation

- Attestation is the activity of making a claim to an appraiser about the properties of a target by supplying evidence which supports that claim
- An attester is a party performing the attestation activity
- An attestation protocol is a cryptographic protocol involving a target an attester, an appraiser and possibly other principals serving as trust proxies
- Target: supply evidence that will be considered authoritative by the appraiser while respecting privacy goals of its target

Remote Attestation

- We denote as remote attestation a protocol whereby a challenge (Chal) verifies the internal state of a device called a prover (Prov)
- This protocol is performed *remotely*, i.e., over the Internet
- Goal: an honest Prov should create an authentication token that convinces Chal that the former is some well-defined expected state
- If Prov has been compromised by an adversary, the authentication token must reflect this

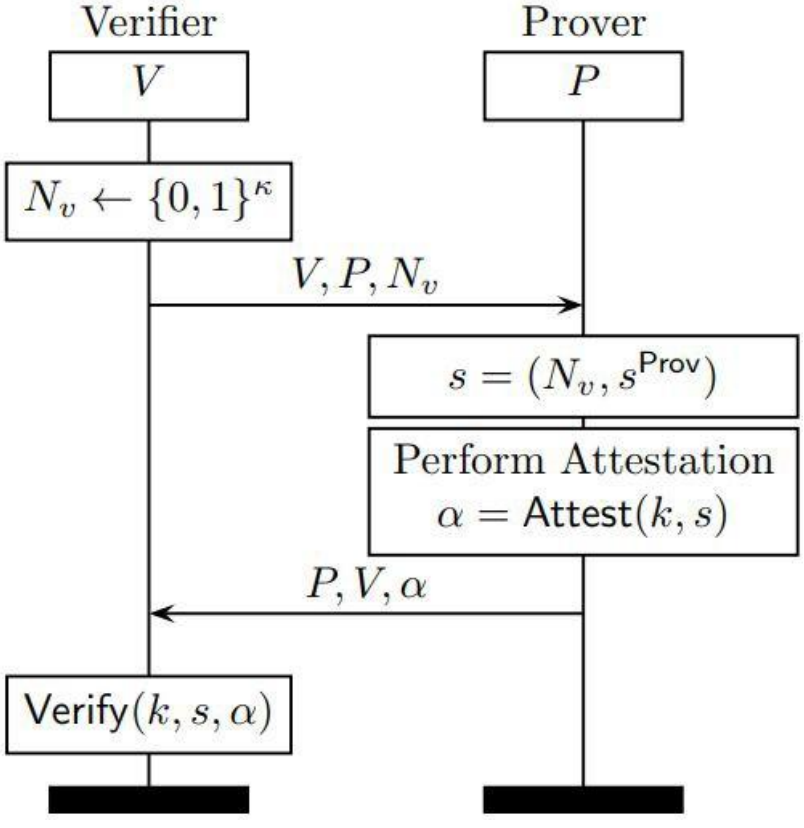
Attestation Protocol

An attestation protocol P comprises the following components

- Steup() -a probabilistic algorithm that, given a security parameter 1^κ outputs a long-term key k;
- Attest(k,.) - a deterministic algorithm that, given a key k and a device state s, outputs an attestation token a
- Verify(k,.,.) - a deterministic algorithm that, given a key k, a device state s, and an attestation token a, outputs 1 iff a corresponds to s, i.e., iff Attest(k,s) = a, and outputs 0 otherwise

Attestation Protocol

- The verifier challenges the prover with a fresh nonce (uniformly random and from a large pool)
- The prover attests its state with the key and creates a token
- The verifier receives the token and decides whether to accept it



Att-Forgery Security

Let us define the following game

Game 1 (Att-Forgery$_{Chal,Prov}$(k)): Chal running P interacts with Prov as follows

1. Chal runs k ← Setup(1^k) and outputs s^{Chal} to Prov
2. Prov is given oracles access to Attest, i.e., adaptively submit q device states and receive the corresponding token
3. Eventually Prov outputs a; the game outputs 1 iff Verify(k,s,a) = 1, i.e., iff a corresponds to s = (s^{Chal},s^{Prov})

Att-Forgery Security

- An honest node has no problem winning the previous game
- If instead Prov has been compromised, its s^{Prov} has changed and must attempt to simulate the operation of Attest

- We can define the following security notion for an attestation protocol

Definition: Att-Forgery Security. An attestation protocol P = (Setup, Attest, Verify) is Att-Forgery-secure if there exists a negligible function *negl* such that for any probabilistic polynomial time prover Prov and sufficiently large k it holds $Pr[\text{Att-Forgery}_{\text{Chal Prov}}(k) = 1] \leq \text{negl}(k)$

System Model

- The central goal of attestation is to verify Prov's state
- Successful execution however does not guarantee that the entire Prov's system can be trusted or that it cannot be compromised after attestation completion
- We assume Prov to be a low end embedded device with a single thread of execution, limited storage capacity, and general complexity
- Although valid for any device, Att-Forgery-security is stronger if the cost of the device is smaller than that of a TPM

System Model

- Prov has the following characteristics
  - *Single memory space* (no separation from kernel and user memory)
  - *Single thread of execution* with exception of interrupts (no direct memory access)
  - Ability to *disable interrupts* and force a region of code to execute automatically
  - Availability of *Read Only Memory* (ROM)
  - Ability to *securely cleanup* (erase) memory upon device reset
  - Hardware-based control mechanism to *prevent unauthorized access* to certain memory location

System Model

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- We make no assumptions about Chal
- A malicious Chal may perform a DoS by forcing Prov to take part in the RA protocol at will
- Malicious Chal does not learn any new information about an honest Prov by performing RA, since Chal must already know the desired state of Prov in order to verify the attestation token

- *We assume that Chal is honest*
- **Note:** Challenger is the term from crypto, verifier is the term from RA. We can use them interchangeably

Adversary Model

- We assume the adversary can compromise Prov at any time
- Once it is compromised, the adversary has control over the *prover* device
- There however needs to be *a key that the adversary cannot access* to although being in control of the prover
- We assume that the adversary cannot modify the hardware components of the compromised device
- We also assume that there is a way to protect Attest against side channel attacks

Properties Required

- Attest needs to have specific security properties
- We assure there exists a secure algorithm to compute *a* based on the prover's state *s* and a prover-specific key k (e.g., via HMAC)
- Attest must satisfy the following security properties
  - Only Attest can compute a valid token *a*
  - *a* accurately captures s, i.e., Attest(k,s) = Attest(k,s') with negligible probability
- Two ways to attack remote attestation
  - Attack 1: The adversary simulates Attest and correctly computes *a*
  - Attack 2: returned *a* does not reflect s, i.e., escape detection

Properties Required

- The key k is the only secret held by Prov, and access to k allows the adversary to simulate Attest (i.e., type 1 attack)

- Exclusive access: attest must have exclusive access to k.
- No leaks: Attest leaks no function of k other than a, i.e., after Attest completes, the entire state of Prov is statistically independent from k
- Immutability: Attest code is immutable. This means that it needs to be executed in-place from immutable memory
- Uninterruptibility: the attacker has no means to interrupt the execution of attest

Features from Properties

- We derive a set of features that are both necessary and sufficient for remote attestation that achieve the five security properties
- **Exclusive access to *k***: in our system model, the best solution is to add a small hardware-based check that monitors the address and the Program Counter (PC) and enforces that *k* is only accessible when PC is within attest
- **No leaks**: we need a way to erase all intermediate values that depend on *k*, except the attestation token *a*, when they are no longer needed

Features from Properties

- **Immutability**: to ensure Attest to be immutable, we place it in ROM and execute it in place. We consider it as an inexpensive way to enforce immutability.
- **Uninterruptibility**: on a platform with a single thread of execution, the adversary can still regain control after invoking Attest by scheduling an interrupt. Both Attest and instructions to enable and disable interrupts should be *atomic*

Features from Properties

- **Invocation from start**: we must enforce exclusive invocation of Attest from its very first instruction. To this aim, custom hardware is needed to enforce the logic: *if the program counter is an address within the Attest code, other than the first instruction address, then the previous instruction must also be within Attest*
- This prevents the adversary to jump in the middle of attest, there is no way to enforce this without OS support
- We hence need to monitor the Attest region and reset the device if detecting illegal behavior

Types of Remote Attestation

- Remote attestation can be performed in several ways, with different requirements in terms of device capabilities, equipment, and security guarantees
- At a high level, we can distinguish between software-based attestation and root of trust-based attestation

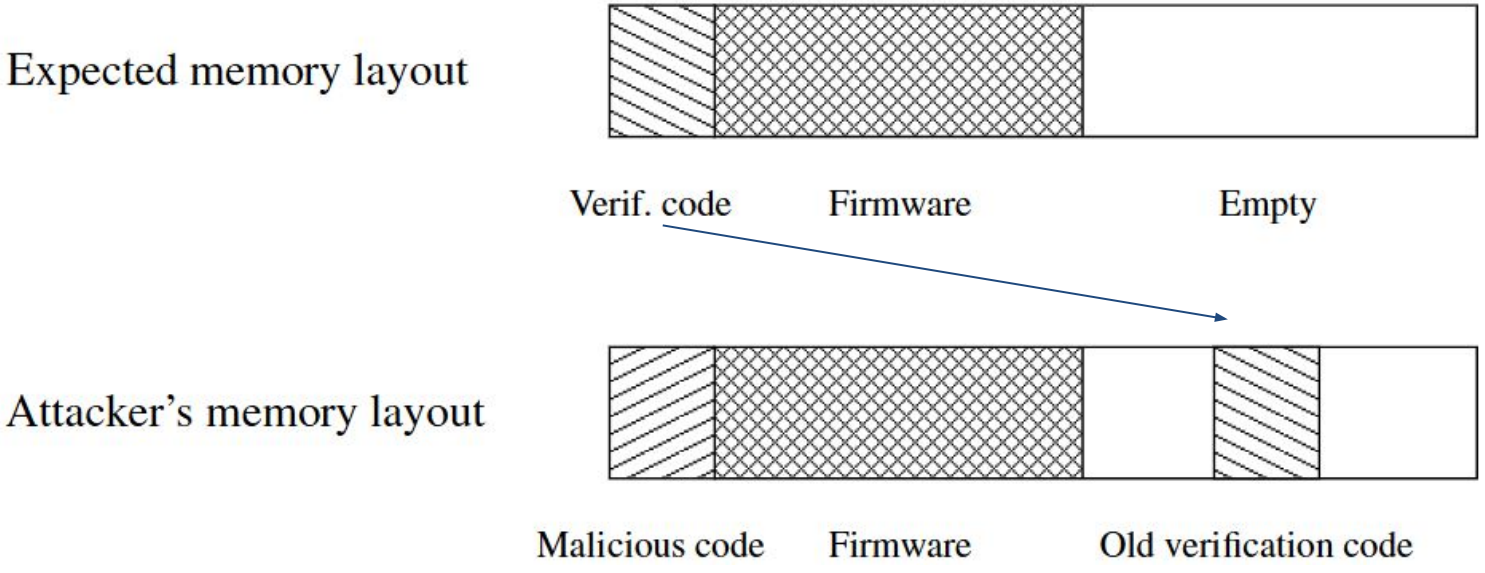
Software-Based Attestation

- In software-based attestation, typically we use timing information to allow the verifier to assess the correctness of the firmware running on the prover
- These approaches generally require strict timing requirements on the network, which might not always be feasible in generic IoT
- They also generally consider a one-hop communication between verifier and prover, which makes it hard to be realized in large IoT networks

Software-Based Attestation

- A Naive approach for verifying the prover's memory content would be to challenge the prover in computing a MAC of the memory content with a verifier-provided key
- The verifier knows the memory content of the legitimate device
- However, an attacker could easily cheat on this
- Indeed, the attacker could save the original memory content and move it to an empty portion of the memory or to an external device that could be accessible when needing to compute the MAC-based proof

Software-Based Attestation



Memory verification attack: The attacker replaces the verification code with malicious verification code and copies the old verification code into empty memory.

Pseudo-Random Memory Traversal

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- The embedded device has a memory-content verification procedure that can be remotely activated by the verifier
- The procedure uses a *pseudorandom memory traversal*
- The verifier sends the device a randomly generated challenge
- The challenge is used as seed for a pseudorandom number generator that generates a list of memory addresses to be checked and iteratively updates a checksum of the memory
- The adversary has no mean to know in advance the portion of the memory that will be checked

Pseudo-Random Memory Traversal

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- In case the verification procedure is heading towards a portion of the memory that has been altered, the attacker needs to divert it to the memory location where the correct copy is stored
- This however causes an increase in the time needed to compute the verification
- Thus the verifier will either see a non valid authentication token or a suspiciously long time needed to compute the authentication token

Desired Properties

- The verification procedure needs the following properties
  - Pseudo-random memory traversal
  - Resistance to precomputation and replay
  - High probability of detecting even single-byte memory changes
  - Small code size
  - Efficient implementation
  - Non-parallelizable

Pseudo-Random Memory Traversal

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- We use a cryptographic pseudo-random number generator to produce the sequence of memory locations to check
- The choice of the generator depends on the CPU architecture
- Helix is a fast stream cipher with built-in MAC functionality optimized for 32-bit architectures
- Its keystream can be used to generate the sequence of memory locations
- Using the challenge as its seed guarantees resistance to pre-computation and replay attacks

High Probability of Detecting Changes

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- **Objective:** small probability that the verification procedure returns the correct checksum when the attacker modifies part of the memory content
- First: we would like the verification procedure to touch every memory location with high probability
- Second: the checksum function should be sensitive to value changes (as small as one byte) and should be difficult for the attacker to find a collusion

High Probability of Detecting Changes

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- First: we would like the verification procedure to touch every memory location with high probability
- *Coupon Collector's Problem:* if each sample of a product is associated with a coupon and there exist n coupons, what is the probability that one needs to buy more than t boxes to get all n coupon types? → O(n log n) accesses to the memory
- Given n total addresses, the number X of memory addresses required is

$$Pr [X > cn \ln n] \leq n^{-c+1}$$

High Probability of Detecting Changes

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Second: the checksum function should be sensitive to value changes (as small as one byte) and should be difficult for the attacker to find a collusion
- For low collision, we need a sufficiently long output: if output is n bits, a lower bound on the collision probability is 2^{-n}
- Let's consider an example on an 8 bit architecture
- *Efficiency* implies that an additional *if* statement (used by the attacker for redirection) introduces substantial slowdown

High Probability of Detecting Changes

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- We use 64 bit checksum and treat the 64-bit checksum as a vector of eighth 8-bit values
- In each iteration, we update one 8-bit value of the checksum incorporating one memory value and mixing it in the RC4 values as well as previous values of the checksum
- Derivation of the 16-bit address of the memory location to be accessed: the high byte of the address is the RC4 value generated in that round, the previous value of the checksum vector is the low byte

Checksum Algorithm

algorithm Verify(m)

//Input: m number of iterations of the verification procedure

//Output: Checksum of memory

Let C be the checksum vector and j be the current index into the checksum vector

for $i \leftarrow 1$ **to** m **do**

//Construct address for memory read

$A_i \leftarrow (RC4_i \ll 8) + C_{((j-1) \mod 8)}$

//Update checksum byte

$C_j \leftarrow C_j + (Mem[A_i] \oplus C_{((j-2) \mod 8)} + RC4_{i-1})$

$C_j \leftarrow rotate\ left\ one\ bit(C_j)$

//Update checksum index

$j \leftarrow (j+1) \mod 8$

return C

Checksum Algorithm

- $A_i \leftarrow (RC4_i << 8) + C_{((i-j) \bmod 8)}$
- The first part left shifts the i-the output of RC4 by 8 bits to create the first eight bits
- Suppose RC4i = 0x12 (00010010), $C_{(j-i) \bmod 8}$ = 0x34 (00110100)
- Then, A_i = (0x12 << 8) + 0x34 = 0x1200 + 0x34 = 0x1234
- With the shifting, $RC4_i$ occupies the upper byte, while C occupies the lower byte
- This results in a more distinct and useful address structure

Checksum Algorithm

- $C_j \leftarrow C_j + Mem[A_i] \text{ XOR } C_{(j-2) \bmod 8} + RC4_{i-1}$
- $C_j \leftarrow$ ***rotate left one bit*** (C_j)
- The checksum accumulates data from memory, current status of checksum vector, and randomness
- XOR ensures non-linearity and hardness for the attacker
- (j-2) mod 8 is a design choice, it ensures historical influence and skips an index to further decouple consecutive bytes
- Rotation spread changes and ensures that small changes in the input have significant changes in the output

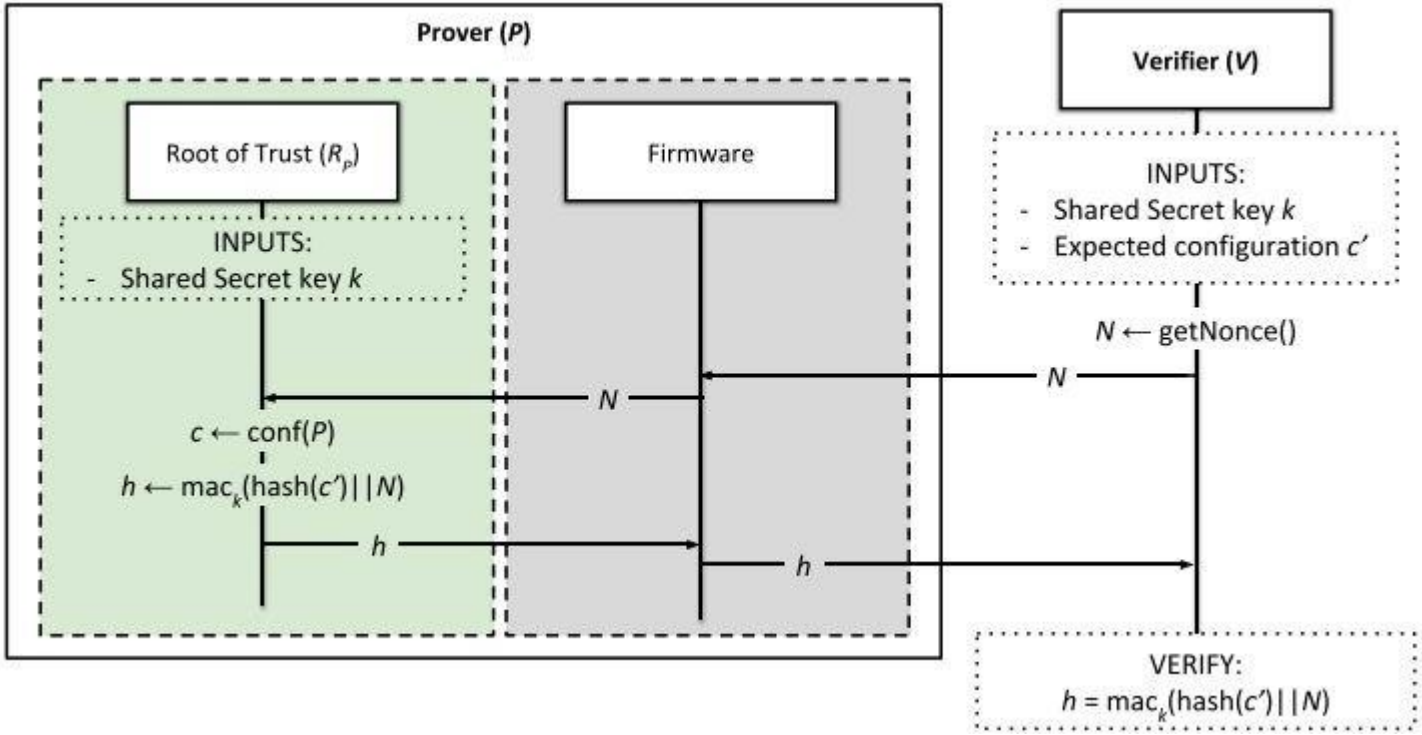
Attestation Based On Root of Trust

- These schemes leverage a root of trust residing inside the prover
- This component is a assumed to be trusted and is the endpoint of the attestation protocol
- It usually comprises a combination of hardware and software
- The value to be attested is stored inside the root of trust
- In IoT devices, the root of trust is realized using hardware with minimal security capabilities, such as code and memory isolation
- Four strategies: interactive RA, Interactive Self-RA, Non-interactive RA, and non-interactive self-RA

Interactive RA

- It consists of an interactive protocol between prover P and verifier V
- V sends a challenge N to P's root of trust R_p, which responds with a proof of the device's configuration, e.g., c = hash(conf(P))||N
- The proof h is either signed (public key crypto) or tagged via MAC (symmetric key)
- V verifies the integrity of P by verifying the authenticity of h

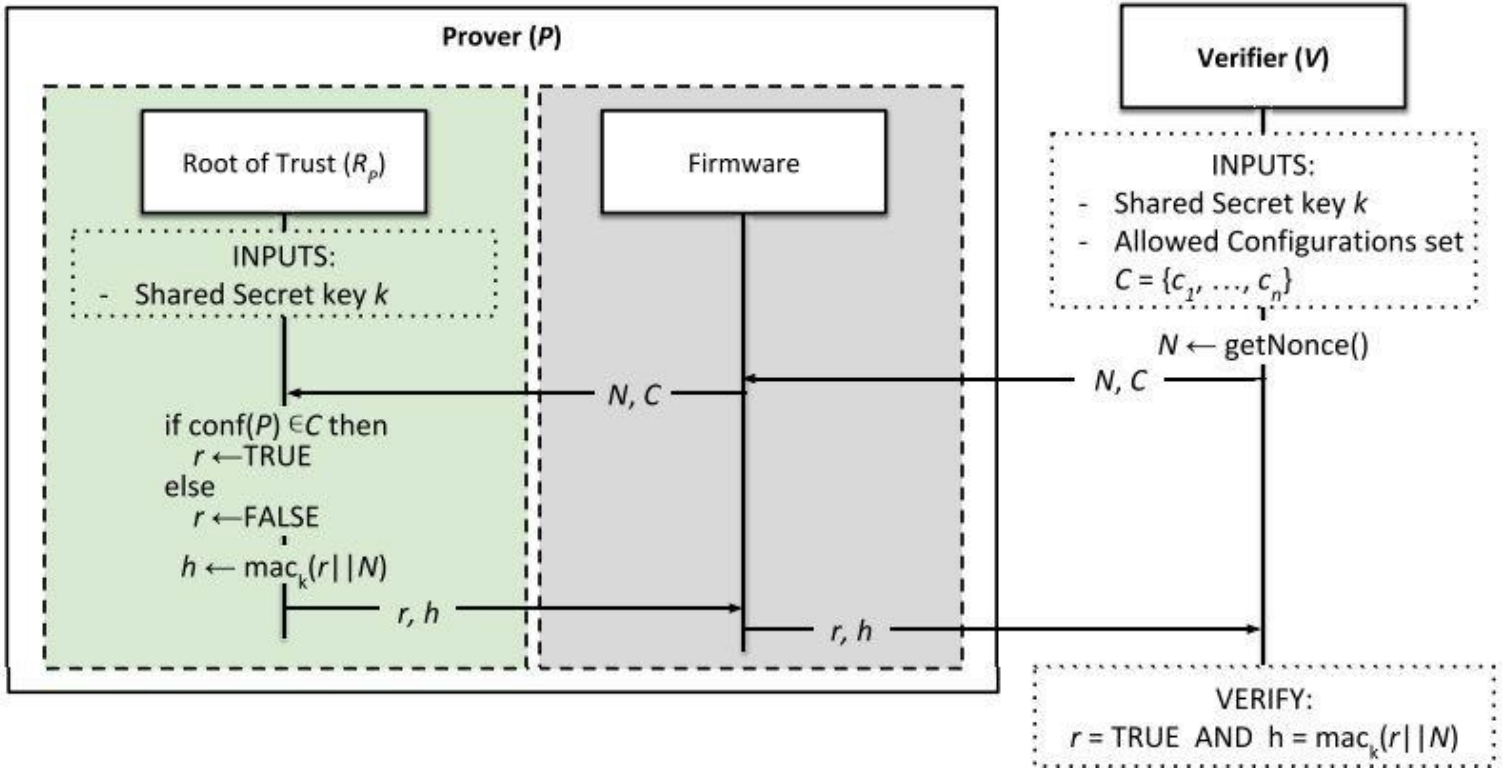
Interactive RA



Interactive Self-RA

- Leveraging the capabilities of Trusted Execution Environments, it is possible to verify c at P's side given the list of potential allowed configurations securely stored and accessible by R_p
- After receiving N from V and computing c, R_p produces a signed/MACed token h authenticating a binary result r (true or false)
- This is then delivered to V as a customized token

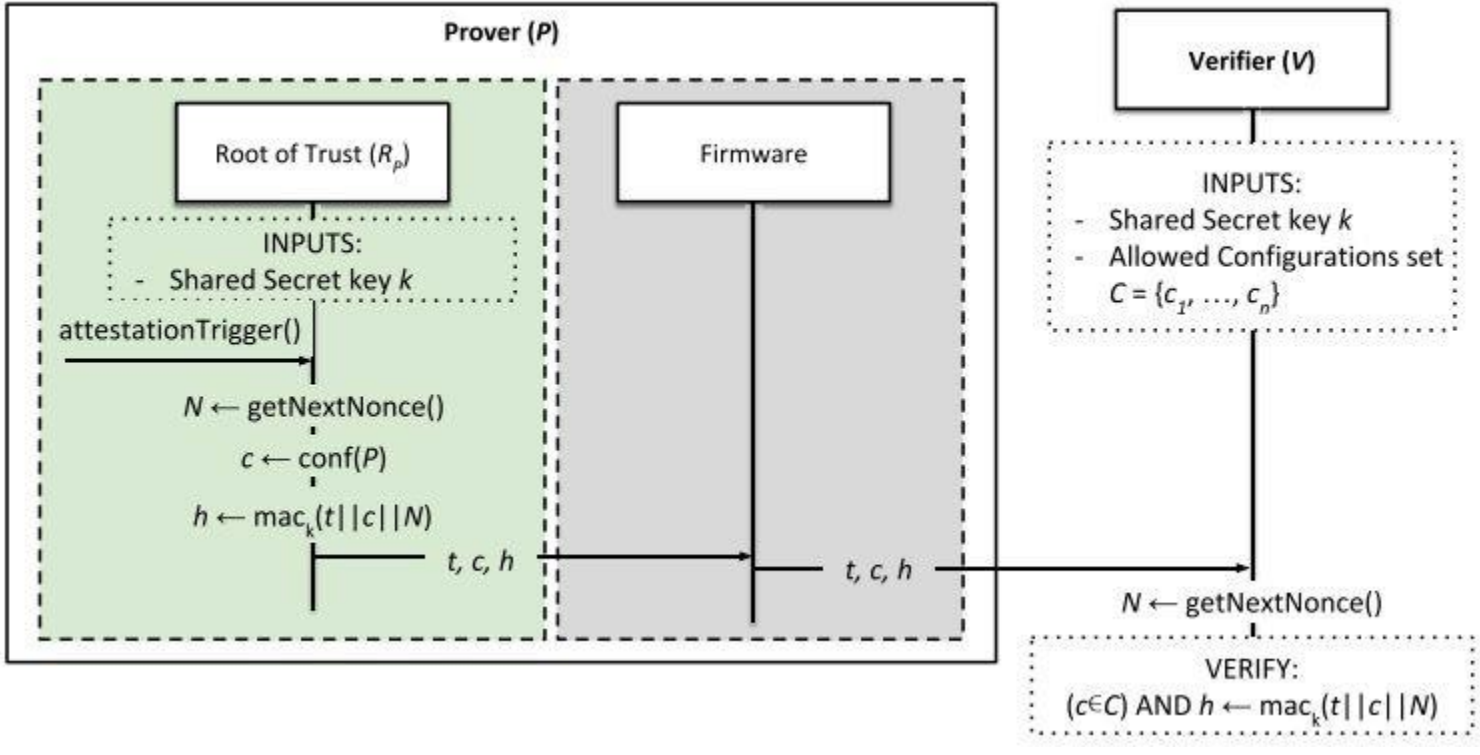
Interactive Self-RA



Non-Interactive RA

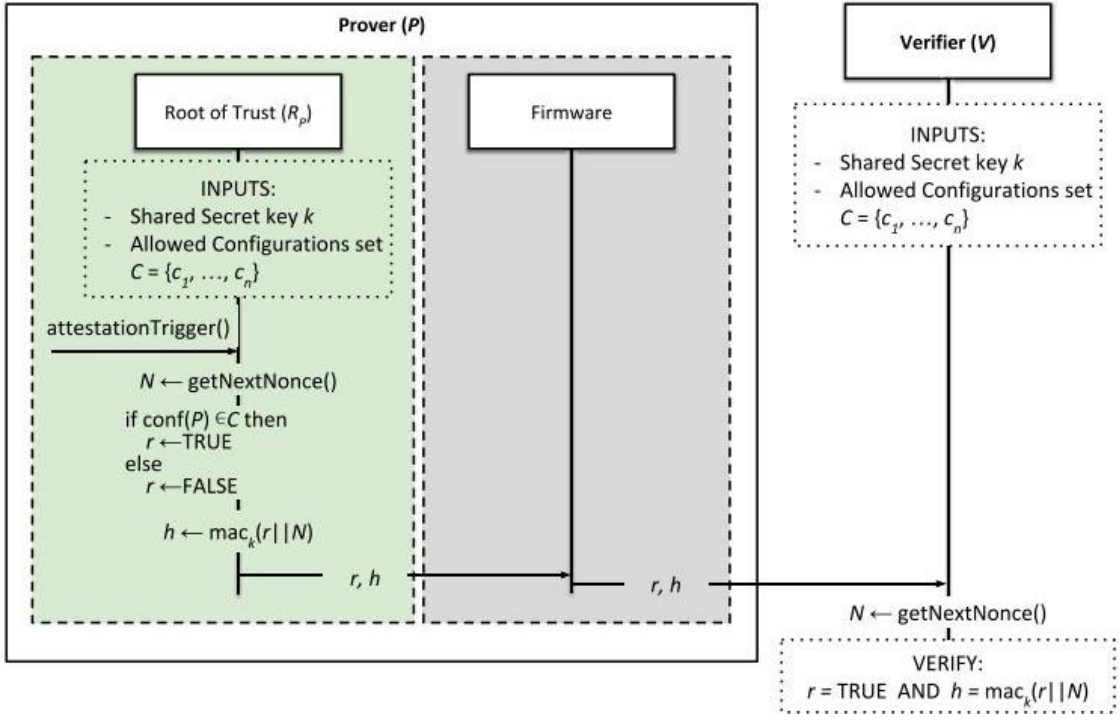
- The prover P autonomously decides the time at which attestation should happen and locally generates a pseudo-random nonce N
- Remove the need for V to start the process, but require additional hardware, e.g., secure source of time
- Examples of additional needs include Real Time Clocks, Attestation Trigger Circuit, or Reliable Read-Only Clocks

Non-Interactive RA



Non-Interactive Self-RA

- Variation of the previous one, where P's TEE know the set of possible configurations and can therefore perform self attestation



Collective RA

- In large networks, it might be convenient to assess the status of multiple nodes instead of performing single attestations
- Collective remote attestation has been introduced to limit the time needed to perform attestation of multiple interconnected devices
- We consider a large network of low-end devices which are heterogeneous in terms of software and hardware configurations
- These are provers, and of course we have the verifier
- We need an additional device, i.e., the aggregator

Formal Verification of RA

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Verifiable Remote Attestation for Simple Embedded Devices (VRASED)
- A hybrid hardware/software solution to provide formal verification
- Security of hardware while minimizing cost thanks to software
- First formally verified remote attestation scheme

Overview of VRASED

- VRASED is composed of a HW module (HW-Mod) and a SW implementation (SW-Att) of Prv's behavior according to the RA protocol
- HW-Mod enforces access control to K (Prv's unique secret key) in addition to secure and atomic execution of SW-Att
- SW-Att is responsible for computing an attestation report

- Computer-aided formal verification involves three basic steps:
  - The system of interest must be described via a formal model (e.g., finite state machine)
  - Properties that the model should satisfy must be formally specified
  - The system model must be checked against formally specified properties to guarantee that the system retains such properties

- Checking can be achieved via either *theorem proving* or *model checking*

- In model checking, properties are specified as formulae using temporal logic and system models are represented as FSMs
- A system is represented by a triple (S, S_0, T) where S is a finite set of states, $S_0 \subseteq S$ is the set of possible initial states, and $T \subseteq S \times S$ is the transition relation set (set of states that can be reached in a single step from each state)
- Thanks to temporal logic we can represent expected system behavior over time
- We use NuSMV

NuSMV

- In NuSMV, properties are specified in Linear Temporal Logic (LTL), particularly useful for sequential systems
- Use of propositional connectives such as conjunction $(\wedge)$, disjunction $(\vee)$, negation $(\neg)$, and implication $(\rightarrow)$
- LTL includes also temporal connectives enabling sequential reasoning
- We define a list of useful temporal connectives

- We define a list of useful temporal connectives
  - **X**ϕ – neXt ϕ: holds if ϕ is true at the next system state.
  - **F**ϕ – Future ϕ: holds if there exists a future state where ϕ is true.
  - **G**ϕ – Globally ϕ: holds if for all future states ϕ is true.
  - ϕ **U** ψ – ϕ Until ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.

- NuSMV works by checking LTL specifications against the system FSM for all reachable states in such FSM

Adv Capability

- We consider an adversary A that can control the entire software state, code, and data of Prv
- A can modify any writable memory and read any memory that is not explicitly protected by access control rules. A can hence read anything that is not protected by HW-Mod
- A can also relocate malware from one memory segment to another to avoid being detected
- A may also have full control over direct memory access controller on Prv

Verification Axioms

- We focus on attestation functionality of Prv
- The verification approach relies on the following axioms
- **A1 - Program counter:** the PC always contains the address of the instruction being executed in a given cycle

- **A2 - Memory address:** whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. R_{en} and W_{en} bits must be set for read and write access, respectively

Verification Axioms

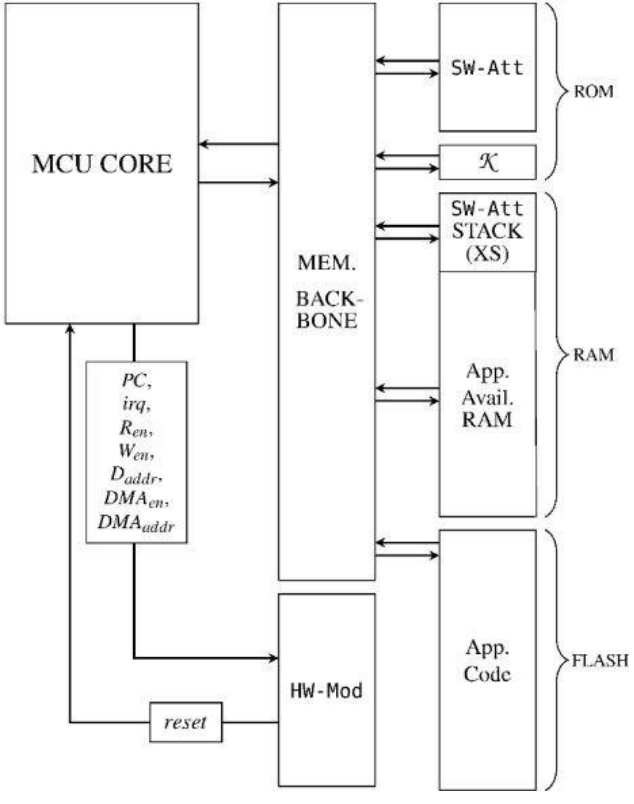
- **A3 - DMA:** whenever a DMA controller attempts to access main system memory, a DMA-address signal DMA_{addr} reflects the address of the memory location being accessed and a DMA_{en} bit must be set. DMA cannot access memory when DMA_{en} is off
- **A4 - MCU reset:** at the end of a successful reset routine, all registers (including PC) are set to zero before resuming normal software execution flow. Since resets are handled by MCU hardware, no way to modify them
- **A5 - Interrupts:** when interrupt, the corresponding irq signal is set

Verification Axioms

- The MSP430 design supports all the five axioms
- Sw-Att uses HACL* HMAC-SHA256 function implemented and verified in F*
- We assume that the standard compiler can be trusted to semantically preserve its expected behavior
- **A6 - Callee-Saves-Register:** any register touched in a function is cleaned by default when the function returns
- **A7 - Semantic preservation:** functional correctness of HMAC semantically preserved when converted in C

VRASED System Architecture

- Implemented by adding HW-Mod to the MCU architecture
- Memory layout is extended to include ROM housing SW-Att code and K
- Access control and SW-Att atomicity are enforced by HW-Mod
- HW-Mod takes 7 input signals from the MCU core



Notation

Notation	Description
PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
irq	Signal that indicates if and interrupt is occurring (1-bit)
CR	(Code ROM) Memory region where SW-Att is stored: $CR = [CR_{min}, CR_{max}]$
KR	($\mathcal{K}$ ROM) Memory region where $\mathcal{K}$ is stored: $KR = [K_{min}, K_{max}]$
XS	(eXclusive Stack) secure RAM region reserved for SW-Att computations: $XS = [XS_{min}, XS_{max}]$
MR	(MAC RAM) RAM region in which SW-Att computation result is written: $MR = [MAC_{addr}, MAC_{addr} + MAC_{size} - 1]$. The same region is also used to pass the attestation challenge as input to SW-Att
AR	(Attested Region) Memory region to be attested. Can be fixed/predefined or specified in an authenticated request from $\mathcal{V}$rf: $AR = [AR_{min}, AR_{max}]$
$reset$	A 1-bit signal that reboots the MCU when set to logic 1
A1, A2, ..., A7	Verification axioms (outlined in section 3.1)
P1, P2, ..., P7	Properties required for secure RA (outlined in section 3.2)

Notation

- We use the following notation
- AR_{min} and AR_{max}: first and last physical addresses of the memory region to be attested
- CR_{min} and CR_{max}: physical addresses of the first and last instructions of SW-Att in ROM
- K_{min} and K_{max}: first and last physical addresses of the ROM region where K is stored
- XS_{min} and XS_{max}: first and last physical addresses of the RAM region reserved for SW-Att computation

Notation

- We use the following notation
- MAC_{addr}: fixed address that stores the result of SW-Att computation (HMAC)
- MAC_{size}: size of HMAC result
- [A,B] denotes that contiguous memory region between A and B
- Therefore, $C \in [A,B] \Leftrightarrow (C \leq B \wedge C \geq A)$
- $PC \in CR$ Holds when PC is within CR_{min} and CR_{max}

FSM Representation

- Each FSM output changes in time as a function of both the current state and current input values
- Each FSM has as inputs a subset of the following signals and wires

$$\{PC, irq, R_{en}, W_{en}, D_{addr}, DMA_{en}, DMA_{addr}\}$$

- Each FSM has only one output, *reset*, that indicates whether any security property was violated. Implicit representation:
  - reset is 1 whenever FSM transitions to reset state
  - reset remains 1 until a transition leaving the reset state is triggered
  - reset is 0 in all other states

RA Soundness

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- RA soundness corresponds to computing an integrity ensuring function over memory at time t
- We use an HMAC computed on memory AR with a one-time key derived from K and Chal
- Since SW-Att is not instantaneous, RA soundness must ensure that attested memory does not change during the computation of the HMAC (temporal consistency)
- In other words, the result of SW-Att call must reflect the entire state of the attested memory at the time when SW-Att is called

RA Soundness

- In other words, the result of SW-Att call must reflect the entire state of the attested memory at the time when SW-Att is called

Definition 1. *End-to-end definition for soundness of* RA *computation*

$$\boldsymbol{G}: \{ PC = CR_{min} \wedge AR = M \wedge MR = \mathcal{C}\text{hal} \wedge [(\neg reset)\ \boldsymbol{U}\ (PC = CR_{max})] \rightarrow$$
$$\boldsymbol{F}: [PC = CR_{max} \wedge MR = HMAC(KDF(\mathcal{K}, \mathcal{C}\text{hal}), M)] \}$$

where M is any AR value and KDF is a secure key derivation function.

Definition 2.

2.1 RA ***Security Game*** *(*RA-*game):*

`Assumptions:`

- `SW-Att` *is immutable, and* $\mathcal{K}$ *is not known to* $\mathcal{A}$
- l *is the security parameter and* $|\mathcal{K}| = |\mathsf{Chal}| = |MR| = l$
- $AR(t)$ *denotes the content in AR at time* t
- $\mathcal{A}$ *can modify AR and MR at will; however, it loses its ability to modify them while* `SW-Att` *is running*

RA-`game:`

1. `Setup:` $\mathcal{A}$ *is given oracle access to* `SW-Att`.
2. `Challenge:` *A random challenge* $\mathsf{Chal} \leftarrow \$\{0,1\}^l$ *is generated and given to* $\mathcal{A}$. $\mathcal{A}$ *continues to have oracle access to* `SW-Att`.
3. `Response:` *Eventually,* $\mathcal{A}$ *responds with a pair* (M, σ), *where* σ *is either forged by* $\mathcal{A}$, *or the result of calling* `SW-Att` *at some arbitrary time* t.
4. $\mathcal{A}$ *wins if and only if* $\sigma = HMAC(KDF(\mathcal{K}, \mathsf{Chal}), M)$ *and* $M \neq AR(t)$.

2.2 RA ***Security Definition:***

An RA *protocol is considered secure if there is no ppt* $\mathcal{A}$, *polynomial in* l, *capable of winning the game defined in 2.1 with* $Pr[\mathcal{A}, \text{RA-}game] > negl(l)$

VRASED SW-Att

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- To minimize required hardware feature, hybrid RA approaches implement integrity ensuring functions (e.g., HMAC) in software
- Derive a new unique context-specific key (key) from the master key K via HMAC-based key derivation function on Chal
- Call HACL*'s HMAC using key as the HMAC key
- Sw-Att resides in ROM, guaranteeing software immutability
- Moreover, HW-Mod enforces that no other software running on Prv can access memory allocated by SW-Att

VRASED SW-Att


```
1   void Hacl_HMAC_SHA2_256_hmac_entry() {
2       uint8_t key[64] = {0};
3       memcpy(key, (uint8_t*) KEY_ADDR, 64);
4       hacl_hmac((uint8_t*) key, (uint8_t*) key, (uint32_t) 64, (uint8_t*)
            CHALL_ADDR, (uint32_t) 32);
5       hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key, (uint32_t) 32, (uint8_t*)
            ATTEST_DATA_ADDR, (uint32_t) ATTEST_SIZE);
6       return();
7   }
```

VRASED SW-Att


```
1   void Hacl_HMAC_SHA2_256_hmac_entry() {
2       uint8_t key[64] = {0};
3       memcpy(key, (uint8_t*) KEY_ADDR, 64);
4       hacl_hmac((uint8_t*) key, (uint8_t*) key, (uint32_t) 64, (uint8_t*)
            CHALL_ADDR, (uint32_t) 32);
5       hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key, (uint32_t) 32, (uint8_t*)
            ATTEST_DATA_ADDR, (uint32_t) ATTEST_SIZE);
6       return();
7   }
```

Memory range to be attested

Definition 3. *SW-Att functional correctness*

$$\boldsymbol{G}: \{ PC = CR_{min} \wedge MR = Chal \wedge [(\neg reset \wedge \neg irq \wedge CR = \texttt{SW-Att} \wedge KR = \mathcal{K} \wedge AR = M)\ \boldsymbol{U}\ PC = CR_{max}]$$
$$\rightarrow \boldsymbol{F}: [PC = CR_{max} \wedge MR = HMAC(KDF(\mathcal{K}, Chal), M)] \}$$

where M is any arbitrary value for AR.

Natural language: if the memory counter is always preserved until the PC gets to CM_{max}, finally MR will contain a valid response

Key Access Control (HW-Mod)

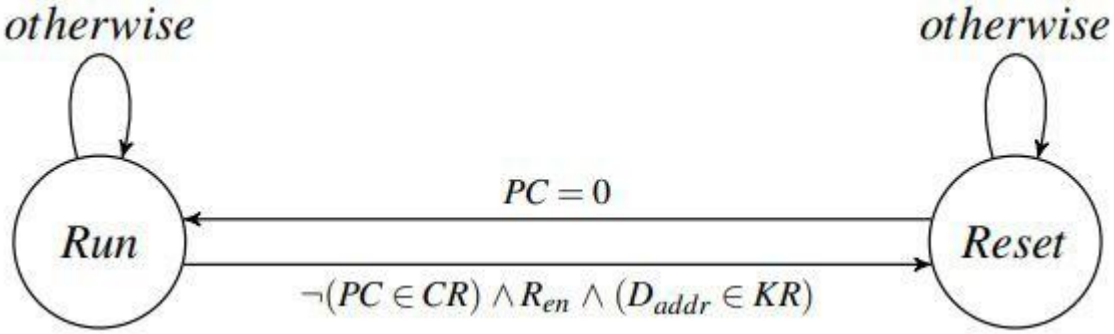
- If malware manages to read K from ROM, it can reply to Vrf with a forged result
- MW-Mod access control (AC) sub-module enforces that K can only be accessed bt SW-Att

$$\mathbf{G}: \{\neg(PC \in CR) \wedge R_{en} \wedge (D_{addr} \in KR) \rightarrow reset\}$$

- The system must transition to the Reset state whenever code from outside CR tries to read from D_{addr} within the key space

Key Access Control (HW-Mod)

- We design a FSM from the LTL specification with two states: *run* and *reset*
- Outputs sìreset =1 when the AC submodule transitions to *reset,* implying a hard reset on the MCU



- We define two LTL specifications for atomic execution and controlled invocation

$$\mathbf{G}: \{[\neg reset \wedge (PC \in CR) \wedge \neg(\mathbf{X}(PC) \in CR)] \rightarrow [PC = CR_{max} \vee \mathbf{X}(reset)]\}$$

$$\mathbf{G}: \{[\neg reset \wedge \neg(PC \in CR) \wedge (\mathbf{X}(PC) \in CR)] \rightarrow [\mathbf{X}(PC) = CR_{min} \vee \mathbf{X}(reset)]\}$$

$$\mathbf{G}: \{irq \wedge (PC \in CR) \rightarrow reset\}$$

- Enforce that the only way for SW-Att execution to terminate is through its last instruction PC=CR_{max}
- Specified by checking current and next PC values using ne**X**t operator
- If current PC value is within SW-Att region and next PC value is out of SW-Att region, then either current PC values is the address of the last instruction in SW-Att(CR_{max}), or reset is triggered in the next cycle

$$\mathbf{G}: \{[\neg reset \wedge (PC \in CR) \wedge \neg(\mathbf{X}(PC) \in CR)] \rightarrow [PC = CR_{max} \vee \mathbf{X}(reset)]\}$$

- Enforce that the only way for PC to enter SW-Att region is through the very first instruction CR_{min}
- This and the previous invariant (LTL specification) imply that it is impossible to jump into the middle of SW-Att or leave SW-Att before reaching the last instruction

$$\mathbf{G}: \{[\neg reset \wedge \neg(PC \in CR) \wedge (\mathbf{X}(PC) \in CR)] \rightarrow [\mathbf{X}(PC) = CR_{min} \vee \mathbf{X}(reset)]\}$$

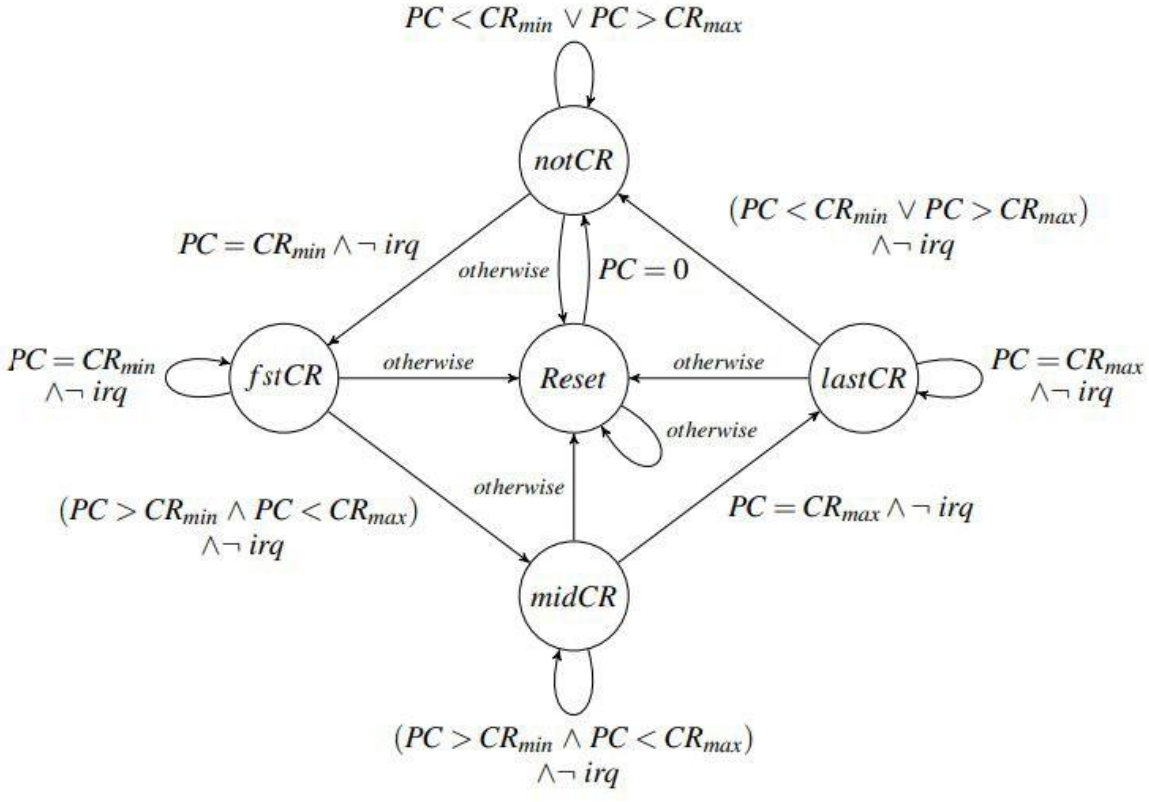
- Atomicy is verified by the following LTL specification
- Although atomicity could be violated by interrupts, this LTL specification prevents an interrupt to happen while SW-Att is executing
- Therefore, if interrupts are not disabled by software running on Prv before calling SW-Att, any interrupt that could violate SW-Att atomicity will necessarily cause an MCU reset

$$\mathbf{G}: \{irq \wedge (PC \in CR) \rightarrow reset\}$$

Verified Model

- The FSM has 5 states
  - Two basic states notCR and midCR represent movements when PC points to an address 1)outside CR, and 2) within CR, respectively
  - Two fstCR and lstCR represent states when the PC points to the first and last instructions of SW-Att, respectively
  - A reset state
- Transition to reset state whenever 1) any sequence of values for PC does not obey the aforementioned conditions, 2) irq is logical 1 while executing SW-Att

Verified Model



Next

- VRASED
- TOCTOU
- CASU
- GAROTA

The TOCTOU Problem in RA

- The scheme we analysed for RA has a fundamental limitation: it measures the state of the prover's executable at time when Sw-Att is executed
- It provides no information about P's state before RA measurements or in between two RA measurements
- Problem of *Time of Check to Time of Use (TOCTOU)*
- This problem is different from temporal consistency among successive RAs

TOCTOU Security Definition

Definition 4.1.

4.1.1 $\mathcal{R}$A-TOCTOU Security Game: Challenger plays the following game with $\mathcal{A}$dv:

(1) Challenger chooses time t_0.
(2) $\mathcal{A}$dv is given full control over $\mathcal{P}$rv software state and oracle access to **Attest** calls.
(3) At time $t_{att} > t_0$, $\mathcal{A}$dv is presented with $Chal$.
(4) $\mathcal{A}$dv wins if and only if it can produce $\mathsf{H}_{\mathcal{A}\mathrm{dv}}$, such that:

$$\textbf{Verify}(\mathsf{H}_{\mathcal{A}\mathrm{dv}}, Chal, M, \cdots) = 1 \tag{5}$$

and

$$\exists_{t_0 \le t_i \le t_{att}} \{AR(t_i) \neq M\} \tag{6}$$

where $AR(t_i)$ denotes the content of AR at time t_i.

4.1.2 $\mathcal{R}$A-TOCTOU Security Definition: An $\mathcal{R}$A scheme is considered TOCTOU-Secure if – for all PPT adversaries $\mathcal{A}$dv – there exists a negligible function negl, such that:

$$Pr[\mathcal{A}\mathrm{dv}, \mathcal{R}\text{A-TOCTOU-game}] \le \mathrm{negl}(l)$$

where l is the security parameter.

- Augmentation of the RA security definition we already saw

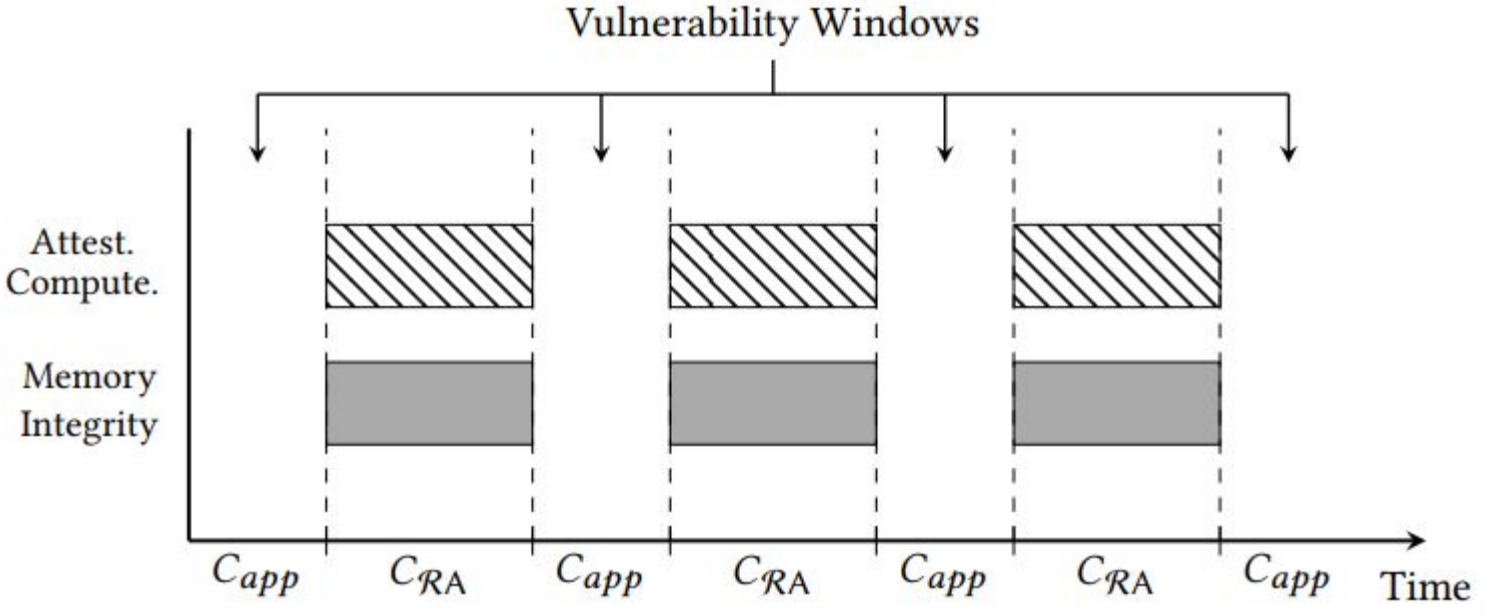
TOCTOU Security Definition

- This definition allow the adversary to success if they produce a valid response even though AR was modified at any point after t_0
- It captures the security against transient attacks where the adversary changes the modified memory back to its expected state before leaving the device
- We only take into account executable memory, and not data memory → common concept for remote attestation

Comparison with Consecutive SM

- Some RA schemes consider consecutive self-measurements to detect transient malware that comes and goes between two successive RA measurements
- Strategy:
  - Prv intermittently and unilaterally invokes its RA functionality
  - Prv either reports to Vrf or accumulates evidences
  - Vrf checks for malware presence in each token

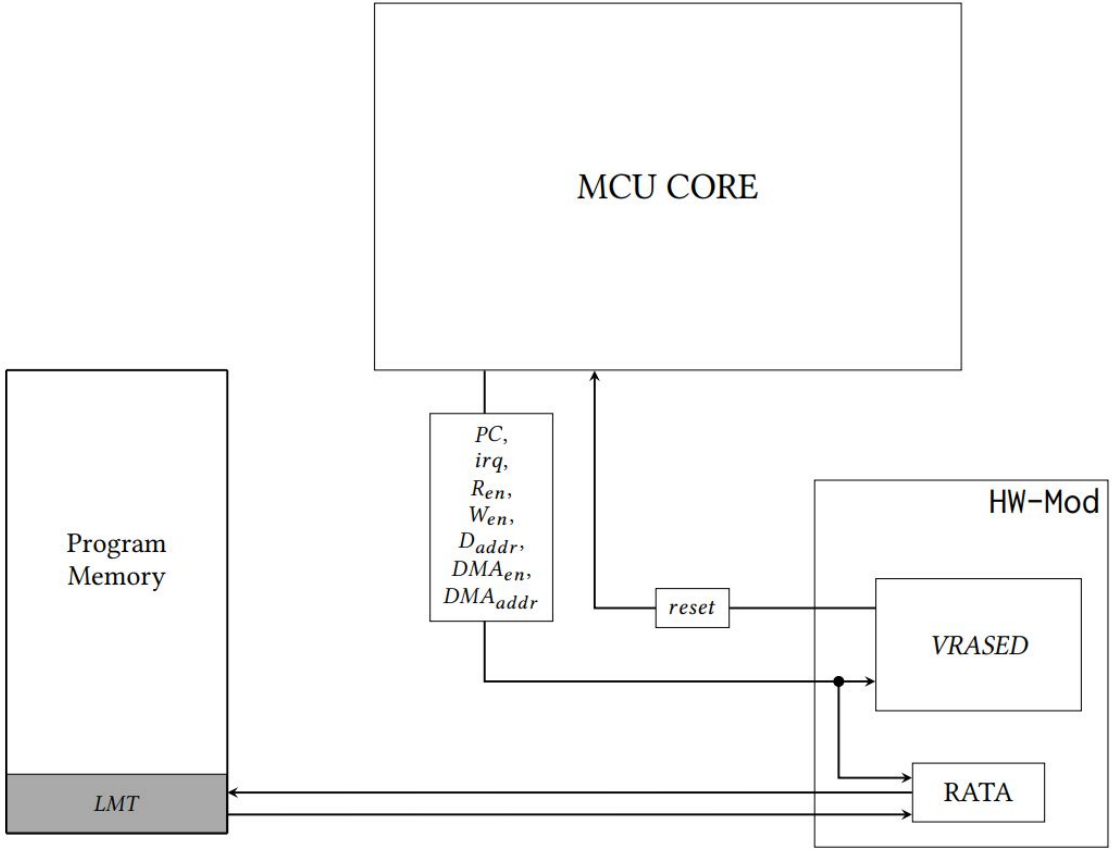
Comparison with Consecutive SM



Comparison with Consecutive SM

- The time between consecutive measurements leaves anyhow space for the attacker
- An idea could be to increase the frequency at which RA is invoked
- However:
  - Determining such high frequency is not trivial
  - The complexity of the scheme is unbearable, leaving the device mostly occupied in the RA task

RATA-A


MCU CORE
PC,
irq,
R_{en},
W_{en},
D_{addr},
DMA_{en},
DMA_{addr}
reset
HW-Mod
VRASED
RATA
Program Memory
LMT

RATA-A

- Monitors a set of CPU signals and detects whenever any location within the attestation region (AR) is written through the signals we saw in VRASED
- Whenever detecting a modification in AR, RATA-A logs the timestamp (latest modification time) obtained from the real-time clock and store it in a fixed memory region
- LMT in AR, and need to enforce that LMT is read-only for all the software executing in the MCU and for DMA

Construction 1 ($RATA_A$). *Let LMT be a memory region within AR ($LMT \in AR$):*

- **Request**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}()$: $\mathcal{V}$rf *generates a random l-bit challenge* Chal $\leftarrow \$\{0,1\}^l$ *and sends it to* $\mathcal{P}$rv.
- **Attest**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}$(Chal): *Upon receiving* Chal, $\mathcal{P}$rv *calls VRASED* SW-Att$\mathcal{R}$A *function to compute* $\mathsf{H} = HMAC(KDF(\mathcal{K}, \text{Chal}), AR)$ *and sends* $t_{LMT}||\mathsf{H}$ *to* $\mathcal{V}$rf, *where t_{LMT} is the value stored in LMT.*
  At all times, $RATA_A$ hardware in $\mathcal{P}$rv *enforces the following invariants:*
  – *LMT is read-only to software:*

  $$\textbf{Formal statement (LTL):} \quad \mathrm{G}\{Mod\_Mem(LMT) \rightarrow reset\} \tag{11}$$

  – *LMT is overwritten with the current time from RTC if, and only if, AR is modified:*

  $$\textbf{Formal statement (LTL):} \quad \mathrm{G}\{Mod\_Mem(AR) \leftrightarrow set_{LMT}\} \tag{12}$$

  where $reset$ is a 1-bit signal that triggers an immediate reset of the MCU, and set_{LMT} is a 1-bit output signal of $RATA_A$ controlling the value of LMT reserved memory. Whenever $set_{LMT} = 1$, LMT is updated with the current value from the real-time clock (RTC). LMT maintains its previous value otherwise.
- **Verify**$^{\mathcal{V}\text{rf}}$(H, Chal, M, t_0, t_{LMT}): t_0 *is an arbitrary time chosen by* $\mathcal{V}$rf, *as in Definition 4.1. Upon receiving* $t_{LMT}||\mathsf{H}$ $\mathcal{V}$rf *checks:*

  $$t_{LMT} < t_0 \tag{13}$$

  $$\mathsf{H} \equiv HMAC(KDF(\mathcal{K}, MR), M) \tag{14}$$

  where M is the expected value of AR reflecting $LMT = t_{LMT}$, as received from $\mathcal{P}$rv. **Verify** *returns* 1 *if and only if both checks succeed.*

- The whole construction can be modeled as a Mealy machine, where the future state depends on the current state and current input values
- Inputs are a set of signals, and outputs are two 1-bit values: reset and set_{LMT} (control the memory location of LMT)
- set_{LMT} is 1 when transitioning to MOD state, 0 otherwise
- It monitors also write access to LMT, and transition to RESET whenever there are attempts to this

- We would like to design a solution that does not rely on the presence of real-time clocks on the prover device
- To this aim, we can leverage the fact that the challenge sent by the verifier is unique on a per-attestation round basis
- We hence introduce RATA-B which is tightly coupled with the authentication of the challenge sent by the verifier

RATA-B

- RATA-B monitors the same set of MCU signals as RATA-A and works by overwriting the special memory region LMT in AR
- Instead of logging the RTC time stamp, it logs the challenge sent by the verifier as part of its request and given as input to the attest()
- LMT is overwritten with the currently received challenge if and only if a modification of AR occurred since the previous attest instance

- Security properties on which RATA-B relies
  - No software running on P can overwrite LMT, which is only modifiable by RATA-B hardware
  - Update to LMT is only triggered immediately after a successful authentication during attest computation
  - The first successful authentication happening after a modification of AR always causes LMT to be update with the current value of Chall stored in MR

RATA-B Verification Mechanism

- Let us denote as $Chal_1$ and H_1 the attestation challenge and response, respectively, received by the verifier
- If H_1 is a valid response (i.e., corresponds to an expected AR value), time t_1 at which such response is received is saved locally by Vrf
- In subsequent attestation results, Vrf checks the value of LMT for correspondence with $Chal_1$
- If LMT different from $Chal_1$, Vrf knows that AR was modified after t_1

RATA-B Observations

- Authentication of Vrf requests is fundamental for RATA-B security
- Without it, the attacker can simply choose a challenge and call Attest after unauthorized modifications of AR, setting LMT=$Chall_{att}$
- Uniqueness of LMT must be enforced via randomly sampling Chal from a sufficiently large space
- If Chal repeats after n requests, the attacker can wait for the n-th authentic request to complete, infect P, and later replay the previous valid response

Construction 2 ($RATA_B$). *Let LMT be a memory region within AR (i.e., $LMT \in AR$) and P be a challenge-time association pair, stored by $\mathcal{V}$rf. Initially $P = (\perp, \perp)$. $RATA_B$ is specified as follows:*

- **Request**$^{\mathcal{V}rf \rightarrow \mathcal{P}rv}$**()**: *$\mathcal{V}$rf generates a pair [Chal, $\mathcal{A}$uth] according to VRASED authentication algorithm (see Appendix A for details) and sends it $\mathcal{P}$rv.*
- **Attest**$^{\mathcal{P}rv \rightarrow \mathcal{V}rf}$**(**Chal, $\mathcal{A}$uth**)**: *Upon receiving [Chal, $\mathcal{A}$uth], $\mathcal{P}$rv behaves as follows:*
  (1) Call VRASED SW-AttRA *function to use $\mathcal{A}$uth to authenticate Chal. If authentication succeeds, proceed to next step. Otherwise, ignore the request.*
  (2) Compute H = $HMAC(KDF(\mathcal{K}, Chal), AR)$, *where* $|LMT| = |Chal|$.
  (3) Send LMT||H to $\mathcal{V}$rf.
  To support this operation, at all times, $RATA_B$ hardware on $\mathcal{P}$rv enforces the following:
  – LMT is read-only to software:

$$\textbf{Formal statement (LTL):} \quad \mathbf{G}\{Mod\_Mem(LMT) \rightarrow reset\} \tag{15}$$

  – LMT is never updated without authentication:

$$\textbf{Formal statement (LTL):} \quad \mathbf{G}\{[\neg UP_{LMT} \wedge \mathbf{X}(UP_{LMT})] \rightarrow \mathbf{X}(PC = CR_{auth})\} \tag{16}$$

  – Modification(s) to AR imply updating LMT in the next authenticated **Attest** *call:*

$$\textbf{Formal statement (LTL):} \quad \mathbf{G}\{Mod\_Mem(AR) \vee reset \rightarrow [(PC = CR_{auth} \rightarrow UP_{LMT})\ \mathbf{W}\ (PC = CR_{max} \vee reset)]\} \tag{17}$$

  where $reset$ is a 1-bit signal that triggers an immediate reset of the MCU, and UP_{LMT} is a 1-bit signal that, when set to 1, replaces the content of LMT with the current value stored in MR region (i.e., Chal). LMT maintains its previous value otherwise.
- **Verify**$^{\mathcal{V}rf}$**(**H, Chal, M, t_0, P, LMT**)**: *Let t_0 denote a time chosen by $\mathcal{V}$rf, as in Definition 4.1. Denote the current values in the challenge-time association pair stored by $\mathcal{V}$rf as $P = (Chal_P, t_P)$. Upon receiving LMT||H, $\mathcal{V}$rf behaves as follows:*
  (1) Check if H ≡ $HMAC(KDF(\mathcal{K}, Chal), M)$, *where M is the expected AR value. Since AR includes LMT, M is set to contain the value of LMT, as received from $\mathcal{P}$rv. Hence, this checks also assures integrity of LMT in AR. If this check fails,* **return 0**, *otherwise, proceed to step 2;*
  (2) If $LMT = Chal_P$ and $t_0 > t_P$, **return 1**, *otherwise, proceed to step 3;*
  (3) Set $P = (LMT, current\_time)$ and **return 0**;

RATA-B FSM

- If a software modification of LMT is attempted, the FSM triggers reset immediately
- If not modification are mare to AR since the previous computation of attest, FSM remains in NotMOD state
- If any modification to AR is detected, FSM transitions to state MOD, indicating that a modification occurred, although not modifying LMT
  → the information to be written in LMT is not available at this time (Chal in next request)

- When a call to attest is made
  - if FSM is in NotMOD, Attest is computed normally and FSM remains in the same state
  - otherwise, FSM stays in MOD state until PC=CR_{auth}, implying successful authentication of Vrf's request. FSM then transitions to UPDATE causing LMT to be overwritten with Chal passed as parameter to the current Attest call

An Example of Collective RA

- Let us consider an IoT distributed system where nodes communicate in an asynchronous manner via a publish/subscribe pattern
- The verifier performs attestation in two steps: initialization at time T_0 and attestation at time T_1
- During initialization time, the verifier initiates the attestation procedure to one or more services (publishers)
- The publisher then performs the local attestation and publishes the result together with the data it produced

An Example of Collective RA

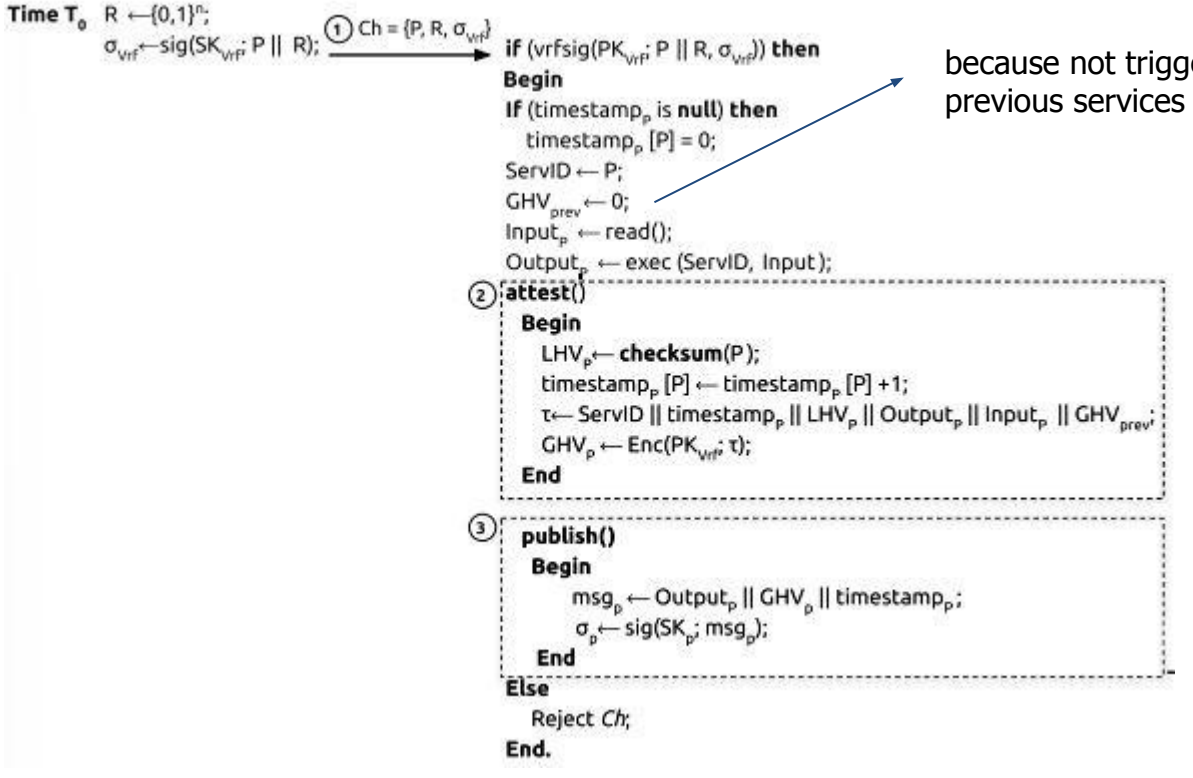
- Every subscriber service that receives the published data also performs attestation
- At attestation time, verifier sends an attestation request to one or more subscriber services, which act as prover for the whole network
- Subscribers report an attestation result that includes the result of all the previous services that were directly or indirectly involved in triggering a given event to which the subscriber was registered

Attestation

- Since it is challenging to synchronize devices' clocks, we leverage the concept of vector clocks
- In this logical model, all clocks are initially set to zero
- Each time a service sends a message it increments its logical clock by one and sends a copy of its own vector
- We assume that each IoT service is composed by a publisher P and a subscriber S

An Example of Collective RA

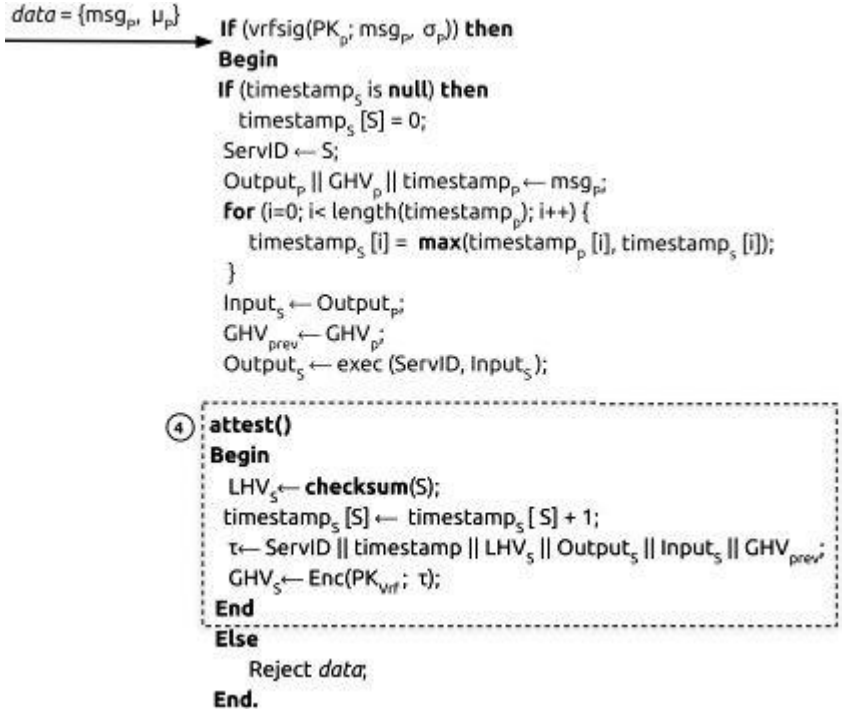
Verifier Publisher



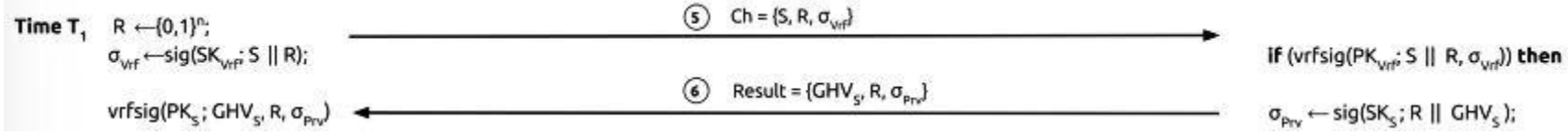
because not triggered by other previous services

An Example of Collective RA

Verifier Publisher Subscriber



An Example of Collective RA



- This is the final attestation procedure, where the verifier retrieves the attestation result GHV_S, singed, and containing the received nonce