On the TOCTOU Problem in Remote Attestation

Ivan De Oliveira Nunes
Rochester Institute of Technology
USA

Sashidhar Jakkamsetti
UC Irvine
USA

Norrathep Rattanavipanon
Prince of Songkla University
Thailand

Gene Tsudik
UC Irvine
USA

ABSTRACT

Much attention has been devoted to verifying software integrity of remote embedded (IoT) devices. Many techniques, with different assumptions and security guarantees, have been proposed under the common umbrella of so-called *Remote Attestation* ($\mathcal{R}$A). Aside from executable's integrity verification, $\mathcal{R}$A serves as a foundation for many security services, such as proofs of memory erasure, system reset, software update, and verification of runtime properties. Prior $\mathcal{R}$A techniques verify the remote device binary at the time when $\mathcal{R}$A functionality is executed, thus providing no information about the device binary before current $\mathcal{R}$A execution or between consecutive $\mathcal{R}$A executions. This implies that presence of transient malware (in the form of modified binary) may be undetected. In other words, if transient malware infects a device (by modifying its binary), performs its nefarious tasks, and erases itself before the next attestation, its temporary presence **will not be detected**. This important problem, called Time-Of-Check-Time-Of-Use (TOCTOU), is well-known in the research literature and remains unaddressed in the context of hybrid $\mathcal{R}$A.

In this work, we propose Remote Attestation with TOCTOU Avoidance (*RATA*): a provably secure approach to address the $\mathcal{R}$A TOCTOU problem. With *RATA*, even malware that erases itself before execution of the next $\mathcal{R}$A, can not hide its ephemeral presence. *RATA* targets hybrid $\mathcal{R}$A architectures, which are aimed at low-end embedded devices. We present two variants – $RATA_A$ and $RATA_B$ – suitable for devices with and without real-time clocks, respectively. Each is shown to be secure and accompanied by a publicly available and formally verified implementation. Our evaluation demonstrates low hardware overhead of both techniques. Compared with current hybrid $\mathcal{R}$A architectures – that offer no TOCTOU protection – *RATA* incurs no extra runtime overhead. In fact, it substantially reduces the time complexity of $\mathcal{R}$A computations: from linear to constant time.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Formal security models**; **Logic and verification**; *Security protocols.*

KEYWORDS

Remote Attestation; TOCTOU attacks; Hardware Security Monitor; Formal Verification; Linear Temporal Logic; Cryptography; Reduction Proofs

ACM Reference Format:
Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. 2021. On the TOCTOU Problem in Remote Attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3460120.3484532

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484532

1 INTRODUCTION

In the last two decades, our society is gradually becoming surrounded by, and dependent upon, a multitude of specialized computing devices that perform a wide range of functions in many aspects of everyday life. They are often referred to as embedded, "smart", CPS or IoT devices, and they vary widely in terms of computational, storage and communication abilities. However, regardless of their purposes and resources, these devices have become popular targets for malicious exploits and malware.

At the low-end of the spectrum, Micro-Controller Units (MCUs) are designed with strict constraints on monetary cost, physical size, and energy consumption. These MCUs exist on the edge of more complex systems, typically interfacing digital and physical domains, and often implementing safety-critical sensing and/or actuation functions. Two prominent examples of such MCUs are: TI MSP430[1] and Atmel ATMega AVR[2]. It is unrealistic to expect such devices, by themselves, to prevent malware infection via sophisticated security mechanisms (similar to those available on laptops, smartphones, or relatively higher-end/higher-power embedded systems[3]). In this landscape, **Remote Attestation ($\mathcal{R}$A)** emerged as an inexpensive and effective means to detect unauthorized modifications to executables of remote low-end devices. Also, $\mathcal{R}$A serves as a foundation for other important security services, such as software updates [1, 2], control-flow integrity verification [3–5], and proofs of remote software execution [6]. Generally speaking, $\mathcal{R}$A allows a trusted entity, called a Verifier ($\mathcal{V}$rf), to ascertain memory integrity of an untrusted remote device, called Prover ($\mathcal{P}$rv). As shown in Figure 1, $\mathcal{R}$A is typically realized as a (deceptively) simple challenge-response protocol:

[1] http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/applications.html
[2] https://www.microchip.com/design-centers/8-bit/avr-mcus
[3] For instance, higher-end devices such as Raspberry Pi, Tessel, and similar, are usually 2-to-3 orders of magnitude more expensive in terms of monetary cost, physical size, and energy consumption than the MCUs targeted in this work.

(1) $\mathcal{V}$rf sends an attestation request with a challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv. This request might also contain a token derived from a secret that allows $\mathcal{P}$rv to authenticate $\mathcal{V}$rf.
(2) $\mathcal{P}$rv receives the request and computes a $\mathcal{C}$hal-based *authenticated integrity check* over a pre-defined memory region. In low-end embedded systems (and in the context of this paper) this region corresponds to the entire executable memory, i.e., program memory. See Section 3.1.
(3) $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.
(4) $\mathcal{V}$rf receives the result and checks whether it corresponds to a valid memory state.

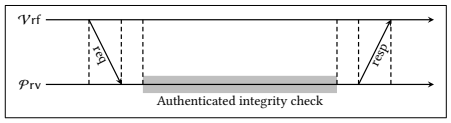


Figure 1: Timeline of a typical $\mathcal{R}$A protocol

The *authenticated integrity check* requires some type of integrity-ensuring function, typically implemented as a Message Authentication Code (MAC), computed over $\mathcal{P}$rv attested memory region. Computing a MAC requires $\mathcal{P}$rv to have a unique secret key, denoted by $\mathcal{K}$– a symmetric key shared with $\mathcal{V}$rf, or a private key for which the corresponding public key is known to $\mathcal{V}$rf. $\mathcal{K}$ must reside in secure storage, and be **inaccessible** to any software running on $\mathcal{P}$rv, except for privileged and immutable attestation code. Since the usual $\mathcal{R}$A threat model assumes a fully compromised software state on $\mathcal{P}$rv, secure storage implies some level of hardware support. Hybrid RA (based on hardware/software co-design) [7–10] is an approach particularly suitable for low-end embedded devices. In hybrid $\mathcal{R}$A designs, the integrity-ensuring function is implemented in software, while hardware controls execution of this software, detecting any violations that might cause unexpected behavior or $\mathcal{K}$ leakage. In a nutshell, hybrid $\mathcal{R}$A provides the same security guarantees as (more expensive) hardware-based $\mathcal{R}$A approaches (e.g., those based on a TPM [11] or other standalone hardware modules), while minimizing modifications to the underlying hardware platform. We overview a concrete hybrid $\mathcal{R}$A architecture in Section 3.2.

Despite much progress, current hybrid $\mathcal{R}$A architectures share a common limitation: they measure the state of $\mathcal{P}$rv executables at the time when $\mathcal{R}$A is executed by $\mathcal{P}$rv. They provide no information about $\mathcal{P}$rv executables **before** $\mathcal{R}$A measurement or its state in **between** two consecutive $\mathcal{R}$A measurements. We refer to this problem as *Time-Of-Check Time-Of-Use* or TOCTOU. While variants of this problem have been discussed before [12–16], it remains unsolved in the context of hybrid $\mathcal{R}$A.

We emphasize that the $\mathcal{R}$A–TOCTOU problem (as formulated in this paper) should not be confused or conflated with the problem of ensuring temporal consistency between attestation and execution of a binary, which is solved by runtime attestation approaches, e.g., [3–6]. Nonetheless, $\mathcal{R}$A of binaries (i.e., static $\mathcal{R}$A) is still relevant in that context because most runtime attestation techniques for low-end devices rely on static $\mathcal{R}$A as a building block (one exception is [17], which, instead, assumes that binaries never change [4]). In fact, as we discuss in Section 8, an $\mathcal{R}$A architecture secure against TOCTOU makes runtime attestation techniques that rely on static $\mathcal{R}$A substantially more efficient.

In practice, the TOCTOU problem leaves devices vulnerable to transient malware which erases itself (its executable) after completing its tasks. This is harmful in settings where numerous MCUs report measurements collected over extended periods. For example, consider several MCU-based sensors that measure energy consumption in a smart city, where large scale erroneous measurement may lead to power outages. If regular $\mathcal{R}$A schemes that are not secure against TOCTOU are used in this case (e.g., by performing $\mathcal{R}$A once a day, or once per billing cycle), security can be subverted by: **(i)** changing sensor software to spoof measurements during regular usage, and **(ii)** reprogramming the sensor back with the expected executable, with both (i) and (ii) occurring in the period between consecutive $\mathcal{R}$A instances. In particular, since the $\mathcal{R}$A request must be received through an *untrusted* communication channel, malware may simply erase itself upon detecting an incoming attestation request, even if the $\mathcal{R}$A schedule is unknown *a priori*. As noted earlier, in settings where detection of runtime violations (e.g., code-reuse and data corruption attacks) is also desirable (e.g., MCU code is written with memory-unsafe languages), TOCTOU-Security of the underlying static $\mathcal{R}$A makes the overall runtime attestation more efficient; see Section 8.

Our approach to mitigating the TOCTOU problem is based on the observation that current hybrid $\mathcal{R}$A techniques use trusted hardware only to detect security violations that compromise execution of $\mathcal{R}$A software itself and take action (e.g., by resetting the device) if such a violation is detected. Whereas, *RATA* main new feature is the use of a minimal (formally verified) hardware component to additionally provide historical context about the state of $\mathcal{P}$rv program memory. This is achieved via secure logging of the latest timing of program memory modifications in a protected memory region that is also covered by $\mathcal{R}$A integrity-ensuring function. This enables $\mathcal{V}$rf to later check authenticity and integrity of $\mathcal{P}$rv memory modifications. This new feature is integrated seamlessly into the underlying $\mathcal{R}$A architecture and the composition is shown to be secure. We believe this results in the following contributions:
– **RA TOCTOU-Security Formulation:** We motivate and formalize TOCTOU in the context of $\mathcal{R}$A. We define $\mathcal{R}$A TOCTOU-Security using a security game (see Definition 4.1) and discuss why current $\mathcal{R}$A techniques based on consecutive self-measurements do not satisfy this definition. We believe our work to be the first formal treatment of this matter. Furthermore, we evaluate practicality of RA techniques based on consecutive self-measurements and argue that using them to obtain TOCTOU-Security incurs extremely high runtime overhead, possibly starving benign applications on $\mathcal{P}$rv.
– **Design, Implementation & Verification:** We propose two techniques: $RATA_A$ and $RATA_B$. The former assumes that $\mathcal{P}$rv has a secure read-only Real-Time Clock (RTC) synchronized with $\mathcal{V}$rf. Since this assumption is unrealistic for many low-end $\mathcal{P}$rv-s, $RATA_B$ trades off the need for a secure RTC for the need to authenticate incoming attestation requests; in fact, this feature is already part of several hybrid $\mathcal{R}$A architectures.

We show that both techniques satisfy the formal definition of TOCTOU-Security, assuming that their implementations adhere to a set of formal specifications, stated in Linear Temporal Logic (LTL).

[4] In many settings involving low-end MCUs this assumption is unrealistic. See Section 2 for details.

Finally, the implementation itself is formally verified to adhere to these LTL specifications, yielding security at both design and implementation levels. Our implementation is publicly available at [18]. It is realized on a real-world low-end MCU (TI MSP430) and is deployed using commodity FPGAs. Experimental results show low hardware overhead, affordable even for cost-sensitive low-end devices.

– **Enhancements to $\mathcal{R}$A and Related Services:** We discuss the implications of *RATA* on $\mathcal{R}$A and related services beyond TOCTOU-Security. In particular, we show that *RATA* can, in most cases, lower $\mathcal{R}$A computational complexity from linear (in terms of attested memory size) to constant time, resulting in significant savings. We also discuss *RATA* benefits for specialized $\mathcal{R}$A applications: (i) real-time systems; (ii) run-time integrity/control-flow attestation; and (iii) collective $\mathcal{R}$A, where a multitude of provers need to be attested simultaneously.

2 SCOPE

This section delineates the scope of this paper in terms of targeted devices and desired security properties.

Low-End Devices: This work focuses on CPS/IoT sensors and actuators (or hybrids thereof) with low computing power. These are some of the smallest and weakest devices based on low-power single-core MCUs with only a few KBytes of program and data memory. Two prominent examples are: Atmel AVR ATmega and TI MSP430: 8- and 16-bit CPUs, typically running at 1-16MHz clock frequencies, with $\approx$ 64 KBytes of addressable memory. SRAM is used as data memory with the size normally ranging between 4 and 16KBytes, while the rest of address space is available for program memory. Such devices usually run software atop "bare metal" and execute instructions in place (physically from program memory) and have no memory management unit (MMU) to support virtual memory.

Our implementation is based on MSP430. This choice is due to public availability of a well-maintained open-source MSP430 hardware design from Open Cores [19]. Nevertheless, our machine model and the entire methodology developed in this paper are applicable to other low-end MCUs in the same class, such as Atmel AVR ATmega. *RATA* main implementation is composed with *VRASED*, a publicly available verified hybrid $\mathcal{R}$A architecture [10], which allows us to demonstrate security. Despite our specific implementation choices, we believe that *RATA* concepts are also applicable to other $\mathcal{R}$A architectures. To support this claim, Appendix D describes *RATA* implementation atop SANCUS [20]: a hardware-based $\mathcal{R}$A architecture also targeting low-end devices. See Section 9 for an overview of various $\mathcal{R}$A architectures.

Detection, Prevention & Memory Immutability: As a detection-oriented security service, $\mathcal{R}$A does not prevent future binary modifications. Therefore, the term TOCTOU should be considered in retrospective. In particular, techniques presented in this paper allow $\mathcal{V}$rf to understand "since when" $\mathcal{P}$rv memory remained the same as reported in the present $\mathcal{R}$A result.

While malware infections can be trivially prevented by making all executable memory read-only (e.g., storing code in ROM), such a drastic approach would sacrifice reconfigurability by making legitimate software updates impossible and would essentially transform the MCU into an Application-Specific Integrated Circuit (ASIC). However, reconfigurability is one of the most important MCU features, perhaps even its entire "raison d'être".

A less drastic approach is to prevent program memory modifications that occur at runtime. This approach is vulnerable to physical attacks, whereby an adversary re-programs $\mathcal{P}$rv directly. More importantly, (even if we rule out physical attacks) it makes remote updates impossible, requiring physical access whenever a device software needs to be updated. Since these devices are often remote or are physically inaccessible (inside a larger system, e.g., a vehicle) low-end MCUs (including aforementioned MSP430 and ATMega) typically do not prevent modifications to program memory. Our detection-based approach conforms with that, allowing changes to binaries and reporting them to $\mathcal{V}$rf: even if they happen between consecutive $\mathcal{R}$A instances. Since $\mathcal{V}$rf is informed about all binary changes on $\mathcal{P}$rv, it can distinguish illegal modifications from expected ones.

3 BACKGROUND & DEFINITIONS

3.1 Device Model & MCU Assumptions

We now overview MCU assumptions relevant to *RATA*. They reflect the behavior of the class of low-end embedded systems discussed in Section 2 and are in accordance with previous work on securing low-end MCUs [1, 6, 7, 10, 21]. In particular, we assume that MCU hardware correctly implements its specifications, as follows:

A1 – *Program Counter (PC):* *PC* always contains the address of the instruction being executed in a given CPU cycle.

A2 – *Memory Address:* Whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, while, for a write access, a data write-enable bit (W_{en}) must be set.

A3 – *DMA:* Whenever the Direct Memory Access (DMA) controller attempts to access the main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and the DMA-enable bit (DMA_{en}) is set. DMA can not access memory without setting DMA_{en}.

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. Resets are handled by the MCU in hardware. Thus, the reset handling routine can not be modified. When a reset happens, the corresponding *reset* signal is set. The same signal is also set when the MCU initializes for the first time.

A5 – *No Data Execution:* Instructions **must** reside (physically) in program memory (PMEM) in order to execute. They are not loaded to DMEM to execute. Data execution is impossible in most low-end devices, including OpenMSP430 used in our prototype. For example, in Harvard-based low-end devices (e.g., AVR Atmega), there is no hardware support to fetch/execute instructions from data memory (DMEM). In other low-end devices that do not prevent data execution by default, this is typically enforced by the underlying hybrid RA architecture. Therefore, even if malware resides in DMEM, it must be copied to, and thus reside in, PMEM before executing.

3.2 $\mathcal{R}$A Definitions, Architectures & Adversarial Model

As discussed in Section 1, $\mathcal{R}$A is typically realized as a challenge-response protocol between $\mathcal{V}$rf and $\mathcal{P}$rv. This notion is captured by a generic syntax for $\mathcal{R}$A protocols in Definition 3.1.

Definition 3.1 (syntax). $\mathcal{R}$A is a tuple (**Request**, **Attest**, **Verify**) of algorithms:

- **Request**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}(\cdots)$: algorithm initiated by $\mathcal{V}$rf to request a measurement of $\mathcal{P}$rv memory range AR (attested range). As part of **Request**, $\mathcal{V}$rf sends a challenge Chal to $\mathcal{P}$rv.
- **Attest**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}(C\text{hal}, \cdots)$: algorithm executed by $\mathcal{P}$rv upon receiving Chal from $\mathcal{V}$rf. Computes an authenticated integrity-ensuring function over AR content. It produces attestation token H, which is returned to $\mathcal{V}$rf, possibly accompanied by auxiliary information to be used by the **Verify** algorithm (see below).
- **Verify**$^{\mathcal{V}\text{rf}}(\text{H}, C\text{hal}, M, \cdots)$: algorithm executed by $\mathcal{V}$rf upon receiving H from $\mathcal{P}$rv. It verifies whether $\mathcal{P}$rv current AR content corresponds to some expected value M (or one of a set of expected values). **Verify** outputs: 1 if H is valid, and 0 otherwise.

Note: In the parameter list, $(\cdots)$ denotes that additional parameters might be included, depending on the specific $\mathcal{R}$A scheme.

Definition 3.1 specifies $\mathcal{R}$A as a tuple (**Request**, **Attest**,**Verify**). **Request** is computed by $\mathcal{V}$rf to produce challenge Chal and send it to $\mathcal{P}$rv. **Attest** is performed by $\mathcal{P}$rv by using Chal to compute an authenticated integrity-ensuring function (e.g., MAC) over attested memory range (denoted by AR) and producing H, which is sent back to $\mathcal{V}$rf for verification. For example, if **Attest** is implemented using a MAC, H is computed as:

$$\text{H} = HMAC(KDF(\mathcal{K}, C\text{hal}), AR) \quad (1)$$

where KDF denotes a key derivation function and $\mathcal{K}$ is a symmetric key shared by $\mathcal{P}$rv and $\mathcal{V}$rf. Upon receiving H, $\mathcal{V}$rf executes algorithm **Verify** by checking if H corresponds to the MAC of some expected value M.

Although techniques discussed in this paper are not tied to a specific $\mathcal{R}$A architecture, we chose to compose *RATA* with *VRASED* [10]. Our choice is motivated by *VRASED* formal security definitions, which allow reasoning about *RATA* secure composition with the underlying $\mathcal{R}$A architecture; see Theorems 5.1 and 6.1. We overview *VRASED* next.

VRASED is a formally verified hybrid RA architecture, based on a hardware/software co-design. It is built as a set of sub-modules, each guaranteeing a specific set of sub-properties. Every sub-module (hardware or software) is individually verified. Finally, composition of all sub-modules is proved to satisfy formal definitions of $\mathcal{R}$A soundness and security. Informally, $\mathcal{R}$A soundness guarantees that an integrity-ensuring function (HMAC in *VRASED*) is correctly computed over attested memory range (AR). It also guarantees that AR can not be modified after the start of $\mathcal{R}$A computation, thus enforcing temporal consistency and protecting against "hide-and-seek" attacks during $\mathcal{R}$A computation [22]. $\mathcal{R}$A security ensures that $\mathcal{R}$A execution generates an unforgeable authenticated memory measurement and that $\mathcal{K}$ used in computing this measurement is not leaked before, during, or after, attestation.

To achieve its aforementioned goals, *VRASED* software part (SW-Att) resides in Read-Only Memory (ROM) and relies on a formally verified HMAC implementation from the HACL* cryptographic library [23]. A typical SW-Att execution proceeds as follows:

(1) Read challenge Chal from a fixed memory region denoted by MR.
(2) Use a Key Derivation Function (KDF) to derive a one-time key from Chal and the attestation master key $\mathcal{K}$: $KDF(\mathcal{K}, MR)$ (where $MR = C\text{hal}$).
(3) **Attest** implementation (SW-Att) generates attestation token H by computing an HMAC over an attested memory region AR using the newly derived key:
$$\text{H} = HMAC(KDF(\mathcal{K}, MR), AR)$$
(4) Overwrite MR with the result H and return execution to unprivileged software, i.e, the normal application(s).

VRASED Hardware (HW-Mod) monitors 7 distinct MCU signals:

- 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 Direct Memory Access (DMA) is currently accessing memory (1-bit);
- DMA_{addr}: Memory address being accessed by DMA.
- irq: Signal that indicates if an interrupt is happening (1-bit);

These signals determine a one-bit *reset* signal output, that, when set to 1, triggers an immediate system-wide MCU reset, i.e., before executing the next instruction. The *reset* output is triggered when *VRASED* hardware detects any violation of security properties. *VRASED* hardware is described in Register Transfer Level (RTL) using Finite State Machines (FSMs). Then, NuSMV Model Checker [24] is used to automatically prove that FSMs achieve claimed security sub-properties. Finally, the proof that the conjunction of hardware and software sub-properties implies end-to-end soundness and security is done using an LTL theorem prover.

Definition 3.2. VRASED Security Game (Adapted from [10])
Notation:
- l is the security parameter and $|\mathcal{K}| = |C\text{hal}| = |MR| = l$
- $AR(t)$ denotes the content of AR at time t

$\mathcal{R}$A-game:
(1) **Setup**: $\mathcal{A}$dv is given oracle access to **Attest** (SW-Att) calls.
(2) **Challenge**: A challenge Chal is generated by calling **Request** (Definition 3.1) and given to $\mathcal{A}$dv.
(3) **Response**: $\mathcal{A}$dv responds with a pair (M, σ), where σ is either a forgery by $\mathcal{A}$dv, or is the result of calling **Attest** (Definition 3.1), at some arbitrary time t.
(4) $\mathcal{A}$dv wins iff $M \neq AR(t)$ and $\sigma = HMAC(KDF(\mathcal{K}, C\text{hal}), M)$.

Note: If, as a part of **Attest**, AR attestation is preceded by a procedure to authenticate $\mathcal{V}$rf, t defined in step 3 is the time immediately after successful authentication, when AR attestation starts.

More formally, *VRASED* end-to-end security proof guarantees that no probabilistic polynomial time (PPT) adversary can win the $\mathcal{R}$A *security game* in Definition 3.2 with non-negligible probability in the security parameter l, i.e., $Pr[\mathcal{A}\text{dv}, \mathcal{R}\text{A-game}] \leq \text{negl}(\text{l})$.

Remark 1: *While aforementioned guarantees ensure consistency of attested memory during attestation computation, VRASED or any prior low-end $\mathcal{R}$A scheme is not TOCTOU-Secure, as modifications*

before attestation remain undetected.

Adversarial Model. We consider a fairly strong adversary $\mathcal{A}$dv that controls the entire software state of $\mathcal{P}$rv, including both code and data. $\mathcal{A}$dv can modify any writable memory and read any memory (including secrets) that is not explicitly protected by trusted hardware. Also, $\mathcal{A}$dv has full access to all DMA controllers, if any are present on $\mathcal{P}$rv. Recall that DMA allows direct access and memory modifications without going through the CPU.

Even though $\mathcal{A}$dv may physically re-program $\mathcal{P}$rv software through wired connection to flash, invasive/tampering hardware attacks are out of scope of this paper: we assume that $\mathcal{A}$dv can not: (1) alter hardware components, (2) modify code in ROM, (3) induce hardware faults, or (4) retrieve $\mathcal{P}$rv secrets via physical side-channels. Protection against physical hardware attacks is orthogonal to our goals and attainable via tamper-resistance techniques [25].

3.3 Linear Temporal Logic (LTL)

Computer-aided formal verification typically involves three basic steps: **First**, the system of interest (e.g., hardware, software, communication protocol) is described using a formal model, e.g., a Finite State Machine (FSM). **Second**, properties that the model should satisfy are formally specified. **Third**, the system model is checked against formally specified properties to guarantee that it retains them. This can be achieved via either Theorem Proving or Model Checking. In this work, we use the latter to verify the implementation of system modules.

In one instantiation of model checking, properties are specified as *formulae* using Linear Temporal Logic (LTL) and system models are represented as FSMs. Hence, 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 – it describes the set of states that can be reached in a single step from each state. The use of LTL to specify properties allows representation of expected system behavior over time.

In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL includes temporal connectives, thus enabling sequential reasoning. In this paper, we are interested in the following temporal connectives:

- $\mathbf{X}\phi$ – neXt ϕ: holds if ϕ is true at the next system state.
- $\mathbf{F}\phi$ – Future ϕ: holds if there exists a future state where ϕ is true.
- $\mathbf{G}\phi$ – 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.
- ϕ **W** ψ – ϕ Weak until ψ: holds if, assuming a future state where ψ holds, ϕ holds for all states prior to that. If ψ never becomes true, ϕ must hold forever. More formally: $\phi \mathbf{W} \psi \equiv (\phi \mathbf{U} \psi) \vee \mathbf{G}(\phi)$

4 $\mathcal{R}$A TOCTOU

This section defines the notion of TOCTOU-Security for $\mathcal{R}$A. We start by formalizing this notion using a security game. Next, we consider the practicality of this problem and overview existing mechanisms, arguing that they do not achieve TOCTOU-Security (neither according to TOCTOU-Security definition, nor in practice) and incur relatively high overhead.

4.1 Notation

We summarize the notation in Table 1.It is mostly consistent with that in *VRASED* [10], with a few additional elements to denote *RATA*-specific memory regions and signals. To simplify the notation, when the value of a given signal (e.g., D_{addr}) is within a certain range (e.g., $AR = [AR_{min}, AR_{max}]$), we write that $D_{addr} \in AR$, i.e.:

$$D_{addr} \in AR \quad \equiv \quad AR_{min} \leq D_{addr} \leq AR_{max} \tag{2}$$

In conformance with axioms discussed in Section 3.1, we use $Mod\_Mem(x)$ to denote a modification to memory address address x. Given our machine model, the following logical equivalence holds:

$$Mod\_Mem(x) \equiv (W_{en} \wedge D_{addr} = x) \vee (DMA_{en} \wedge DMA_{addr} = x) \tag{3}$$

this captures the fact that a memory modification can be caused by either the CPU (reflected in signals $W_{en} = 1$ and $D_{addr} = x$) or by the DMA (signals $DMA_{en} = 1$ and $DMA_{addr} = x$). We also use this notation to represent a modification to a location within a contiguous memory region R as:

$$Mod\_Mem(R) \equiv (W_{en} \wedge D_{addr} \in R) \vee (DMA_{en} \wedge DMA_{addr} \in R) \tag{4}$$

Table 1: Notation

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 accessing memory (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
irq	Signal that indicates if an interrupt is happening
CR	Memory region where SW-Att is stored: $CR = [CR_{min}, CR_{max}]$
MR	(MAC Region) Memory region in which SW-Att computation result is written: $MR = [MR_{min}, MR_{max}]$. The same region is also used to pass the attestation challenge as input to SW-Att
AR	(Attested Region) Memory region to be attested. Corresponds to all executable memory (program memory) in the MCU: $AR = [AR_{min}, AR_{max}]$
LMT	(Latest Modification Time) Memory region that stores a timestamp/challenge corresponding to the last AR modification
CR_{Auth}	The first instruction in *VRASED* SW-Att that is executed after successful authentication of $\mathcal{V}$rf request.
set_{LMT}	($RATA_A$) A 1-bit signal overwrites LMT with the current RTC time, when set to logical 1.
UP_{LMT}	($RATA_B$) A 1-bit signal overwrites LMT with the content of MR when set to logical 1.

4.2 TOCTOU-Security Definition

Definition 4.1 captures the notion of TOCTOU-Security. In it, the game formalizes the threat model discussed in Section 3.2, where $\mathcal{A}$dv controls $\mathcal{P}$rv entire software state, including the ability to invoke **Attest** at will. The game starts with the challenger ($\mathcal{V}$rf) choosing a time t_0. At a later time (t_{att}), $\mathcal{A}$dv receives Chal and wins the game if it can produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ that is accepted by **Verify** as a valid response for expected AR value M, when, in fact, there was a time between t_0 and t_{att} when $AR \neq M$.

This definition augments $\mathcal{R}$A security (Definition 3.2) to incorporate TOCTOU attacks, by allowing $\mathcal{A}$dv to succeed if it produces the expected response, even though AR was modified at any point after t_0, where t_0 is chosen by $\mathcal{V}$rf. For example, if $\mathcal{V}$rf wants to know if AR remained in a valid state for the past two hours, $\mathcal{V}$rf chooses

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 $\mathrm{H}_{\mathcal{A}\mathrm{dv}}$, such that:

$$\textbf{Verify}(\mathrm{H}_{\mathcal{A}\mathrm{dv}}, C\mathrm{hal}, M, \cdots) = 1 \quad (5)$$

and

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

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.

t_0 as $t_0 = t_{att} - 2h$. Note that this definition also captures security against transient attacks wherein $\mathcal{A}$dv changes modified memory back to its expected state and leaves the device, thus attempting to hide its ephemeral modification from the upcoming attestation request. This attack is undetectable by all $\mathcal{R}$A schemes that are not TOCTOU-Secure.

Remark 2: *Recall that AR corresponds to the executable part of $\mathcal{P}$rv memory, i.e., program memory. Since data memory is not executable (see Section 3.1), changes to data memory are not taken into account by Definition 4.1. RATA relation to runtime/data-memory attacks is discussed in Section 8.4.*

4.3 TOCTOU-Secure $\mathcal{R}$A vs. Consecutive Self-Measurements

$\mathcal{R}$A schemes based on consecutive self-measurements [12, 26] attempt to detect transient malware that comes and goes between two successive $\mathcal{R}$A measurements. The strategy is for $\mathcal{P}$rv to intermittently (based on an either periodic or unpredictable schedule) and unilaterally invoke its $\mathcal{R}$A functionality. Then, either $\mathcal{P}$rv self-reports to $\mathcal{V}$rf [26], or it accumulates measurements locally and waits for $\mathcal{V}$rf to explicitly request them [12]. Upon receiving $\mathcal{R}$A response(s), $\mathcal{V}$rf checks for malware presence at the time of each $\mathcal{R}$A measurement. Time intervals used in these $\mathcal{R}$A schemes are depicted in Figure 2.

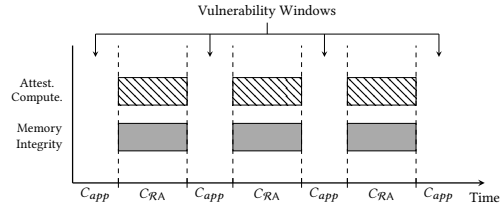


Figure 2: Consecutive Self-Measurements

Clearly, all self-measument schemes always leave some time between consecutive $\mathcal{R}$A instances, during which transient malware presence would not be detected. The only way to detect all transient malware with self-measurement schemes is to invoke $\mathcal{R}$A functionality on $\mathcal{P}$rv with a sufficiently high frequency, such that the fastest possible transient malware can not come and go undetected. However, even if it were easy (which it is not) to determine such "sufficiently high frequency", doing so would be horrendously costly, as we show below. We define CPU utilization (U) in a consecutive scheme as the percentage of CPU cycles that can be used by a regular application (C_{app}), i.e, cycles other than those spent on self-measurements ($C_{\mathcal{RA}}$):

$$U = \frac{C_{app}}{C_{app} + C_{\mathcal{RA}}} \quad (7)$$

As discussed above, guaranteed detection of transient malware via consecutive self-measurements requires:

$$C_{app} < C_{\mathcal{A}\mathrm{dv}} \quad (8)$$

where $C_{\mathcal{A}\mathrm{dv}}$ is the hypothetical number of instruction cycles used by the fastest transient malware that can infect $\mathcal{P}$rv, perform its tasks, and erase itself. To illustrate this point, we assume a conservative value for $C_{\mathcal{A}\mathrm{dv}} = 10^6$ cycles, in which case:

$$C_{\mathcal{A}\mathrm{dv}} = 10^6 \implies C_{app} < 10^6 \implies U < \frac{10^6}{10^6 + C_{\mathcal{RA}}} \quad (9)$$

For example, with $C_{\mathcal{RA}}$, consider the number of CPU cycles required by *VRASED* (other hybrid $\mathcal{R}$A architectures, e.g., [7], have similar costs) to attest a program memory of 4KB: $C_{\mathcal{RA}} = 3.6 \times 10^6$ CPU cycles (about half a second in a typical 8MHz low-end MCU).

$$U < \frac{10^6}{10^6 + 3.6 \times 10^6} \implies U < 21.74\% \quad (10)$$

To detect transient malware, a large fraction of CPU cycles (almost 80% in this toy example) is spent on $\mathcal{R}$A computation. In practice, it is hard to determine $C_{\mathcal{A}\mathrm{dv}}$ and, in some cases (e.g., changing a general-purpose input/output value to trigger actuation), it is likely to be much less than 10^6 cycles, resulting in even lower CPU utilization left for legitimate applications running on $\mathcal{P}$rv. Therefore, detection of all transient malware using consecutive self-measurements is impractical. This also applies to the case where the interval between successive measurements is variable and/or randomly selected from a range $[0, t_{max}]$. As discussed in [26], this is because it must be that $t_{max} < C_{\mathcal{A}\mathrm{dv}}$ in order to achieve negligible probability of malware evasion.

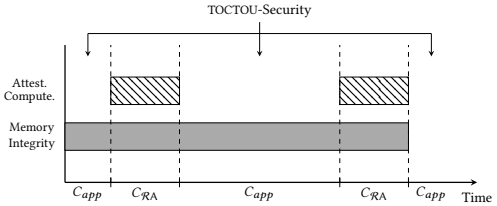


Figure 3: TOCTOU-Secure $\mathcal{R}$A

As shown in Figure 3, TOCTOU-Secure $\mathcal{R}$A (per Definition 4.1) allows $\mathcal{V}$rf to ascertain memory integrity independently from the time between successive $\mathcal{R}$A measurements, regardless of transient malware speed. In the next sections, we propose two TOCTOU-Secure techniques and show their security with respect to Definition 4.1.

5 $RATA_A$: RTC-BASED TOCTOU-SECURE TECHNIQUE

In hybrid $\mathcal{R}$A, trusted software (SW-Att) is usually responsible for generating the authenticated $\mathcal{R}$A response (H) and all semantic information therein. Meanwhile, trusted hardware (HW-Mod) is responsible for: (i) ensuring that SW-Att executes as expected, (ii) preventing leakage of its cryptographic secrets, and (iii) handling unexpected or malicious behavior during execution. To address TOCTOU, we propose a paradigm shift by allowing (formally verified) HW-Mod to also provide some context about $\mathcal{P}$rv memory state.

We now overview $RATA_A$, a simple technique that requires $\mathcal{P}$rv to have a reliable read-only Real-Time Clock (RTC) synchronized with $\mathcal{V}$rf. However, RTCs are not readily available on low-end MCUs and secure clock synchronization in distributed systems is challenging [27–29], especially for low-end devices [30, 31]. Nonetheless, we start with this simple approach to show the main idea behind TOCTOU-Secure $\mathcal{R}$A. Next, Section 6 proposes an alternative variant that removes the RTC requirement, as long as $\mathcal{V}$rf requests are authenticated by $\mathcal{P}$rv. (Note that $\mathcal{V}$rf authentication is already part of some hybrid $\mathcal{R}$A architectures, including *VRASED*.)

5.1 $RATA_A$: Design & Security

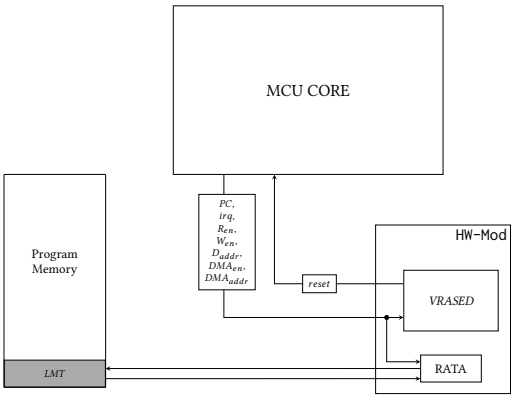


Figure 4: *RATA* **module in the overall system architecture**

$RATA_A$ is illustrated in Figure 4; it is designed as a verified hardware module behaving as follows:

(1) It monitors a set of CPU signals and detects whenever any location within *AR* is written. This is achieved by checking the value of signals D_{addr}, W_{en}, DMA_{addr}, and DMA_{en} (see Section 3.2). These signals allow detection of memory modifications either by CPU or DMA.

(2) Whenever a modification in *AR* is detected, $RATA_A$ logs the timestamp by reading the current time from the RTC and storing it in a fixed memory location, called Latest Modification Time (*LMT*).

(3) In the memory layout, $LMT \in AR$. Also, $RATA_A$ enforces that LMT is always read-only for all software executing on the MCU, and for DMA.

Note that, by enforcing $LMT \in AR$, the attestation result $H = HMAC(KDF(\mathcal{K}, MR), AR)$ includes the authenticated value of *LMT* – the time corresponding to the latest modification of *AR*. As part of the **Verify** algorithm, $\mathcal{V}$rf compares this information with the time of the last authorized modification (t_0 of Definition 4.1) of *AR* to check whether any unauthorized modifications occurred since then. The general idea is further specified in Construction 1, which shows how $RATA_A$ can be seamlessly integrated into *VRASED*, enforcing two additional properties in hardware to obtain TOCTOU-Security. These properties are formalized in LTL in Equations 11 and 12 of Construction 1.

We show that Construction 1 is secure as long as $RATA_A$ implementation adheres to LTL statements in Equations 11 and 12. This verification is discussed in Section 5.2. The cryptographic proof is by reduction from *VRASED* security (per Definition 3.2) to TOCTOU-Security (per Definition 4.1) of Construction 1. For its part, *VRASED* is shown secure according to Definition 3.2 as long as HMAC is a secure, i.e., existentially unforgeable [32], MAC (see [10] for details). The proof of Theorem 5.1 is presented in Appendix B.

THEOREM 5.1. *Construction 1 is TOCTOU-Secure according to Definition 4.1 as long as VRASED is secure according to Definition 3.2.*

5.2 $RATA_A$: Implementation & Verification

Construction 1 (and respective security proof) assumes that properties in Equations 11 and 12 are enforced by $RATA_A$. Figure 5 shows a formally verified FSM corresponding to this implementation. It enforces two properties of Equations 11 and 12. This FSM is implemented as a Mealy machine, where output changes anytime based on both the current state and current input values. The FSM takes as an input a subset of signals, shown in Figure 4, and produces two 1-bit outputs: *reset* to trigger an immediate reset and set_{LMT} to control the value of *LMT* memory location (see Construction 1). *reset* is 1 whenever FSM transitions to *RESET* state and while it remains in that state; it remains 0 otherwise. Whereas, set_{LMT} is 1 when FSM transitions to *MOD* state, and becomes 0 whenever it transitions out of *MOD* state. $set_{LMT} = 0$ in all other cases.

The FSM works by monitoring write access to *LMT* and transitioning to *RESET* whenever such attempt happens. When the system is running (i.e., $reset = 0$), FSM also monitors write access to *AR* and transitions to *MOD* state whenever it happens. The FSM transitions back to *NotMOD* state if *AR* is not being modified. We design the FSM in Verilog HDL and automatically translate into SMV using Verilog2SMV [33]. Finally, we use NuSMV model checker [24] to prove that the FSM complies with invariants 11 and 12. The implementation and correspondent verification are available in [18].

Remark 3: *Since deletion is a "write" operation, malware can not erase itself at runtime without being detected by RATA. Conversely, any attempt to reprogram flash (AR) directly via wired connection requires device re-initialization. Both $RATA_A$/$RATA_B$ always update LMT on initialization/reset/reboot. Hence, these modifications are also detected.*

Remark 4: *The ability to cause a reset by attempting to write to LMT yields no advantage for $\mathcal{A}$dv, since any bare-metal software (including malware) can always trigger a reset on an unmodified low-end device, e.g., by inducing software faults.*

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

- **Request**$^{\mathcal{V}\text{rf}\to\mathcal{P}\text{rv}}()$: *$\mathcal{V}$rf generates a random l-bit challenge $C\text{hal} \leftarrow \$\{0,1\}^l$ and sends it to $\mathcal{P}$rv.*
- **Attest**$^{\mathcal{P}\text{rv}\to\mathcal{V}\text{rf}}(C\text{hal})$: *Upon receiving Chal, $\mathcal{P}$rv calls VRASED* SW-Att *function to compute $\mathsf{H} = HMAC(KDF(\mathcal{K}, C\text{hal}), AR)$ 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:
  $$\textit{Formal statement (LTL):} \quad \mathrm{G}\{Mod\_Mem(LMT) \rightarrow reset\} \tag{11}$$
  – LMT is overwritten with the current time from RTC if, and only if, AR is modified:
  $$\textit{Formal statement (LTL):} \quad \mathrm{G}\{Mod\_Mem(AR) \leftrightarrow set_{LMT}\} \tag{12}$$
  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**$^{\mathcal{V}\text{rf}}(\mathsf{H}, C\text{hal}, M, t_0, t_{LMT})$: *$t_0$ 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.*

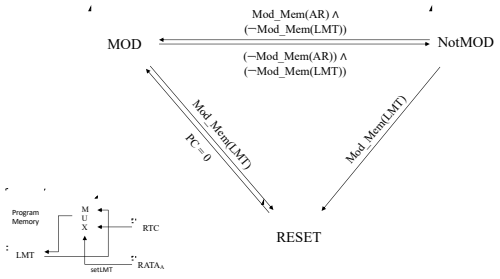


Figure 5: $RATA_A$ FSM for RTC-based TOCTOU-secure $\mathcal{R}$A

6 $RATA_B$: CLOCKLESS TOCTOU-SECURE $\mathcal{R}$A TECHNIQUE

We now describe $RATA_B$: a TOCTOU-Secure technique that requires no clock on $\mathcal{P}$rv. We apply the ideas from $RATA_A$ by using hardware to convey authenticated information about *the time of the latest memory modification* as part of the attestation result. However, lack of RTC precludes any notion of "time" on $\mathcal{P}$rv. To cope with this, we rely on $\mathcal{V}$rf to convey information tied to a given point in time, according to $\mathcal{V}$rf own local clock. This is done as a part of $\mathcal{R}$A **Request** algorithm. In fact, $RATA_B$ uses the attestation challenge (Chal) itself in this task, taking advantage of the fact that Chal is *unique* per **Request** and is available in any $\mathcal{R}$A technique, thus incurring no additional communication overhead. Security of $RATA_B$ is tightly coupled with authentication of $\mathcal{V}$rf **Request**, which is already part of *VRASED* architecture [10]; see Appendix A for details.

6.1 $RATA_B$ – Design & Security

The design of $RATA_B$ remains consistent with Figure 4. $RATA_B$ monitors the same set of MCU signals as $RATA_A$ and also works by overwriting the special memory region $LMT \in AR$. However, instead of logging an RTC timestamp to LMT, it logs Chal, which was sent by $\mathcal{V}$rf as a part of its **Request** and given as input to **Attest**(Chal, ...). LMT is overwritten with the currently received Chal if and only if, a modification of AR occurred since the previous **Attest** instance. In summary, $RATA_B$ security relies on the following properties, enforced by its verified hardware implementation (see Section 6.2):

(1) Similar to $RATA_A$, no software running on $\mathcal{P}$rv can overwrite LMT, i.e., LMT is only modifiable by $RATA_B$ hardware.

(2) An update to LMT is triggered only immediately after a successful authentication during **Attest** computation.

(3) The first successful authentication happening after a modification of AR always causes LMT to be updated with the current value of Chal which is stored in MR. (Recall from Table 1 that MR is the memory location from which **Attest** reads the value of Chal.)

Let $C\text{hal}_1$ and H_1 denote the attestation challenge and response successfully sent/received by $\mathcal{V}$rf, in a given $\mathcal{R}$A interaction. $\mathcal{V}$rf interprets $\mathcal{R}$A results as follows: if H_1 is a valid response, i.e., it corresponds to an expected AR value, time t_1 when such response is received is saved locally by $\mathcal{V}$rf, associated to $C\text{hal}_1$. In subsequent attestation results (H_2, H_3, ...), $\mathcal{V}$rf checks the value of LMT for correspondence with $C\text{hal}_1$. If $LMT \neq C\text{hal}_1$, $\mathcal{V}$rf learns that AR was modified after t_1. This stems from $RATA_B$ verified module, which guarantees that LMT is always overwritten with the newly received challenge if a TOCTOU happens between consecutive calls to **Attest**. In this design, we highlight the following observations:

– **Authentication of $\mathcal{V}$rf Request** is instrumental to $RATA_B$ security. Without it, $\mathcal{A}$dv can simply choose $C\text{hal}_{\mathcal{A}\text{dv}}$ and call **Attest**($C\text{hal}_{\mathcal{A}\text{dv}}$) after an unauthorized modification of AR, thus setting $LMT = C\text{hal}_{\mathcal{A}\text{dv}}$ of its choice. By choosing $C\text{hal}_{\mathcal{A}\text{dv}}$ as a value previously used by $\mathcal{V}$rf, $\mathcal{A}$dv can easily convince $\mathcal{V}$rf that no TOCTOU occurred between measurements. In other words, lack of **Request** authentication allows $\mathcal{A}$dv to modify LMT at will, rendering write protection of LMT useless.

– **Uniqueness of** LMT must be enforced, e.g., by having $\mathcal{V}$rf randomly sample Chal from a sufficiently large space or use Chal as a monotonically increasing counter, depending on specifics of **Request** algorithm. If Chal is reused after n instances of **Request**, $\mathcal{A}$dv can wait for the n-th authentic **Request** to complete, infect $\mathcal{P}$rv, perform its tasks, and leave $\mathcal{P}$rv before the $(n+1)$-st **Request** occurs (with a reused Chal), resulting in a

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**$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}$**()**: *$\mathcal{V}$rf generates a pair [$C$hal, $\mathcal{A}$uth] according to VRASED authentication algorithm (see Appendix A for details) and sends it $\mathcal{P}$rv.*
- **Attest**$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}$**(**$C$hal, $\mathcal{A}$uth**)**: *Upon receiving [Chal, $\mathcal{A}$uth], $\mathcal{P}$rv behaves as follows:*
  (1) Call VRASED SW-Att$\mathcal{R}$A *function to use $\mathcal{A}$uth to authenticate Chal. If authentication succeeds, proceed to next step. Otherwise, ignore the request.*
  (2) Compute H $= HMAC(KDF(\mathcal{K}, C\text{hal}), AR)$, *where $|LMT| = |C\text{hal}|$.*
  (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:
  $$\textit{Formal statement (LTL):} \quad \mathbf{G}\{Mod\_Mem(LMT) \rightarrow reset\} \tag{15}$$
  – LMT is never updated without authentication:
  $$\textit{Formal statement (LTL):} \quad \mathbf{G}\{[\neg UP_{LMT} \wedge \mathbf{X}(UP_{LMT})] \rightarrow \mathbf{X}(PC = CR_{auth})\} \tag{16}$$
  – Modification(s) to AR imply updating LMT in the next authenticated **Attest** *call:*
  $$\textit{Formal statement (LTL):} \quad \mathbf{G}\{Mod\_Mem(AR) \vee reset \rightarrow [(PC = CR_{auth} \rightarrow UP_{LMT})\ \mathbf{W}\ (PC = CR_{max} \vee reset)]\} \tag{17}$$
  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**$^{\mathcal{V}\text{rf}}$**(**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 $P = (C\text{hal}_P, t_P)$. Upon receiving LMT||H, $\mathcal{V}$rf behaves as follows:*
  (1) Check if H $\equiv HMAC(KDF(\mathcal{K}, C\text{hal}), M)$, *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 $LMT = C\text{hal}_P$ and $t_0 > t_P$, **return 1**, *otherwise, proceed to step 3;*
  (3) Set $P = (LMT, current\_time)$ and **return 0**;

valid response and compromised TOCTOU-Security. For example, if we use LMT as a dirty-bit (instead of Chal), security can be subverted in two **Request**-s, even if they are properly authenticated.

$RATA_B$ is specified in Construction 2. $\mathcal{P}$rv hardware module controls the value of a 1-bit signal UP_{LMT}. When set to 1, UP_{LMT} updates LMT with the current value of MR; otherwise, LMT maintains its current value. $RATA_B$ hardware detects successful authentication of $\mathcal{V}$rf by checking whether the program counter PC points to the instruction reached immediately after successful authentication. Note that the instruction at location CR_{auth} is never reached **unless** authentication succeeds. Unlike $RATA_A$, $\mathcal{V}$rf in $RATA_B$ learns whether a modification occurred since a previous successful attestation response, though not the exact time of that modification. $RATA_B$ security is stated in Theorem 6.1 and the proof is deferred to Appendix C.

Theorem 6.1. *Construction 2 is TOCTOU-Secure according to Definition 4.1 as long as VRASED is secure according to Definition 3.2.*

6.2 $RATA_B$: Implementation & Verification

Proof of Theorem 6.1 assumes that $RATA_B$ hardware adheres to properties in Equations 15 to 17. Figure 6 shows $RATA_B$ implementation as an FSM formally verified to adhere to these properties. It takes as input a subset of signals, shown in Figure 4 and outputs two 1-bit signals: $reset$ triggers an immediate system-wide reset and UP_{LMT} controls updates to LMT region. $UP_{LMT} = 1$ whenever the FSM transitions to state $UPDATE$ and has value 0 in all other states. $reset = 1$ whenever the FSM transitions to state $RESET$ and remains unchanged while in this state; it remains 0 otherwise. The FSM operates as follows:

(1) If a software modification of LMT is attempted, FSM triggers $reset$ immediately, regardless of what state it is in.

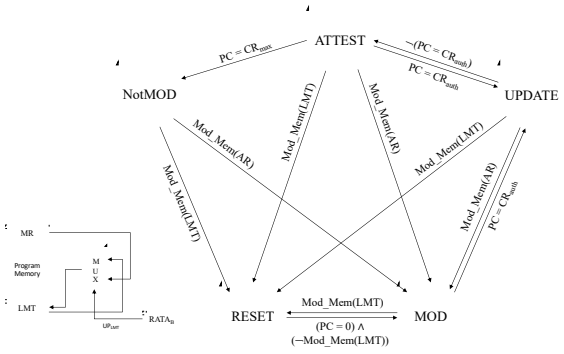


Figure 6: $RATA_B$ FSM for clock-less TOCTOU-secure $\mathcal{R}$A

(2) If no modifications are made to AR since the previous computation of **Attest**, FSM remains in $NotMOD$ state.
(3) At any point in time, if a modification to AR is detected, FSM transitions to state MOD. This transition indicates that a modification occurred, although it neither alters any output, nor modifies LMT. This is because the information to be written to LMT (the value of Chal in the next **Request**) is not available at this time.
(4) When a call to **Attest** is made, two possible actions can occur:
  (a) If FSM is in $NotMOD$ state, **Attest** is computed normally and FSM remains in the same state.
  (b) Otherwise, FSM stays in MOD state until condition $PC = CR_{auth}$ is met, implying successful authentication of $\mathcal{V}$rf **Request**. Then, FSM transitions to state $UPDATE$ causing UP_{LMT} to be set during the transition. Hence, LMT is overwritten with Chal passed as a parameter to the current **Attest** call. Note that update to LMT happens before the computation of the integrity-ensuring function (HMAC) over AR, which happens in state $ATTEST$.

Therefore, attestation result H will reflect $LMT = Chal$ as part of AR. Once **Attest** is completed ($PC = CR_{max}$), FSM transitions back to $NotMOD$.

The same verification tool-chain discussed in Section 5.2 is used to prove that this FSM adheres to LTL statements in Equations 15, 16, and 17.

7 EVALUATION

Our prototype is built upon a representative of the low-end class of devices – TI MSP430 MCU family [34]. It extends *VRASED* (itself built atop OpenMSP430 [19] – an open-source implementation of MSP430) to enable TOCTOU detection. It is synthesized and executed using Basys3 commodity FPGA prototyping board.

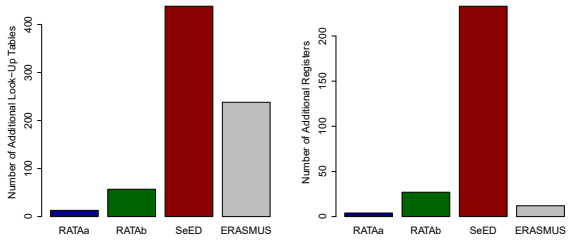
Hardware Overhead. Table 2 reflects the analysis of *RATA* verified hardware overhead. Similar to some related work [1, 5, 6, 10, 35, 36], we consider the hardware overhead in terms of additional LUTs and registers. The increase in the number of LUTs can be used as an estimate of the additional chip cost and size required for combinatorial logic, while the number of extra registers offers an estimate on state registers required by sequential logic in *RATA* FSMs. Compared to *VRASED*, the verified implementation of $RATA_A$ module takes 4 additional registers and 13 additional LUTs, while $RATA_B$ increases the number of LUTs and registers by 57 and 27, respectively. As far as the unmodified OpenMSP430 architecture, this represents the overhead of 1.4% LUTs and 1.4% registers for $RATA_A$ and 3.8% LUTs and 4.8% registers for $RATA_B$.

Architecture	Hardware		Verification		
	LUT	Reg	Verified LoC	Time (s)	Memory (MB)
OpenMSP430	1849	692	-	-	-
VRASED	1862	698	474	0.4	13.6
$RATA_A$	1875	702	601	0.6	19.7
$RATA_B$	1919	725	656	0.8	26.1

Table 2: Additional hardware and verification cost

Runtime Overhead. *RATA* does not require any modification to $\mathcal{R}$A execution. It only ensures that information about the latest modification of attested memory is factored into the attestation result. Hence, it incurs no extra runtime cycles or additional RAM allocation, on top of that of *VRASED* architecture. In fact, as we discuss next, in Section 8, **Attest** runtime can be reduced to the time to attest only LMT. The runtime reduction is presented in Figure 8. This represents a reduction of ≈ 10 times compared, e.g., to the number of cycles to attest an AR of size 4KBytes. The runtime savings increase linearly with the size of AR.

Memory Overhead. $RATA_A$ requires 128-bit of additional storage: 64 bits for RTC and 64 bits for LMT. RTC is implemented using a 64-bit memory cell incremented at every clock cycle. This guarantees that RTC does not wrap around during $\mathcal{P}$rv lifetime since it would take more than 70, 000 years for that to happen on MSP430 running at 8MHz and incrementing RTC at every cycle. In $RATA_A$, LMT is implemented as a 64-bit memory storage and updates its content with RTC value whenever set_{LMT} bit is on. For $RATA_B$, the memory overhead increases to a total of 512 bits. 256 bits of memory are required by the implementation of *VRASED* authentication module, while another 256 bits are used to implement LMT that updates its content with $Chal$ when applicable (as described in Section 6). This small reserved memory corresponds to 0.1% of MSP430 memory address space (64KBytes in total).



(a) Additional LUTs (b) Additional Registers

Figure 7: Hardware overhead. Comparison between *RATA* and techniques based on self-measurements.

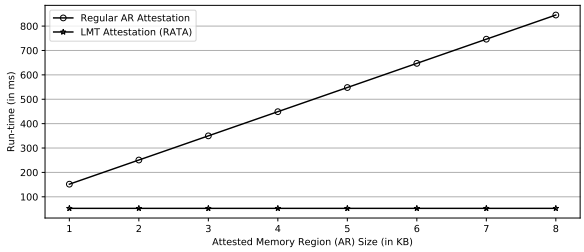


Figure 8: Comparison of LMT attestation time Case-1) with regular attestation of AR (Case-2), as a function of $|AR|$. $|LMT|$ is 32 Bytes. Results on the MSP430 MCU running at 8MHz.

Verification resources. We verify *RATA* on an Ubuntu 18.04 machine running at 3.40GHz. Results are shown in Table 2. $RATA_A$ adds 127 lines of verified Verilog code on top of *VRASED*. These are needed to enforce 2 invariants in Equations 11 and 12. $RATA_B$ incurs 182 additional lines of verified Verilog code, needed to enforce the 3 invariants in Equations 15, 16, and 17. Besides that, *RATA* verification requires checking existing *VRASED* invariants. Overall verification process takes less than one second and consumes at most 26MB of memory.

Comparison. We compare *RATA* hardware overhead with that of two recent self-measurement $\mathcal{R}$A techniques: SeED [26] and ERASMUS [12]. Even though, as discussed in Section 4.3, these techniques do not achieve TOCTOU-Security (per Definition 4.1), we believe that they are the most closely related approaches to *RATA*. SeED extends a 32-bit Intel architecture, which is higher-end than our target devices, i.e., a 16-bit TI MSP430. Whereas, ERASMUS was implemented on MSP430. Figure 7 compares *RATA* to SeED and ERASMUS in terms of numbers of additional LUTs and registers. $RATA_A$ require fewer LUTs, compared to both SeED and ERASMUS. Whereas, $RATA_B$ necessitates more registers, compared to ERASMUS, it uses less LUTs than both self-measurements techniques. In summary, both *RATA*-s incur low overhead: $< 5\%$ increase for both LUTs and registers.

8 USING *RATA* TO ENHANCE $\mathcal{R}$A & RELATED SERVICES

We now discuss how *RATA* can make $\mathcal{R}$A and related services simpler and more efficient.

8.1 Constant-Time $\mathcal{R}$A

One notable and beneficial feature of *RATA* is that, most of the time, $\mathcal{R}$A no longer needs to be computed over the entire *AR*, which significantly reduces $\mathcal{R}$A execution time on $\mathcal{P}$rv.

If $\mathcal{V}$rf already knows *AR* contents from a previous attestation result, it suffices to show that *AR* was not changed since then. This can be done by attesting *LMT* **by itself**, instead of *AR* in its entirety, resulting in substantial reduction of computation time from linear in the size of *AR* to constant: $|LMT|$, i.e., 32 bytes. As such, $\mathcal{R}$A is performed differently, in two possible cases:

– **Case-1:** if no modification to *AR* happened since the last attestation (denoted by t_{att}), call **Attest** on *LMT* region only. **Verify** checks for $\mathsf{H} \equiv HMAC(KDF(\mathcal{K}, \mathcal{C}\text{hal}), LMT)$. $\mathcal{V}$rf then learns whether *AR* was modified since the previous measurement, solely based on *LMT*. By checking that *LMT* corresponds to $t_0 < t_{att}$, this result confirms that *AR* remained the same in the interim. Therefore, measuring *AR* again is unnecessary and doing so would be redundant.

– **Case-2:** If *AR* was modified since the last attestation, call **Attest** covering entire *AR*. **Verify** is computed normally as described in Constructions 1 or 2, depending on the implementation, i.e., $RATA_A$ or $RATA_B$.

Remark 5: *Note that $\mathcal{P}$rv$\mathcal{R}$A functionality can easily detect whether AR was modified (in order to decide between attesting with* ***Case-1*** *or* ***Case-2****) by checking the value of LMT, which is readable (though not writable) in software.*

Most of the time, $\mathcal{P}$rv is expected to be in a benign state (i.e., no malware), especially if $\mathcal{A}$dv knows that its presence is guaranteed to be detectable. In such times, size of attested memory can be reduced from several KBytes (e.g., when *AR* is the entire program memory on a low-end $\mathcal{P}$rv) to a mere 32 Bytes (*LMT* size), Figure 8 depicts an empirical result on the MSP430 MCU showing how this optimization can significantly reduce $\mathcal{R}$A runtime overhead.

In the rest of this section, we discuss some implications of this optimization, along with security improvements offered by *RATA*, to different branches of $\mathcal{R}$A and related security services.

8.2 Atomicity & Real-Time Settings

Security of hybrid $\mathcal{R}$A architectures generally depends on *temporal consistency* of attested memory. Simply put, temporal consistency means "no modifications to *AR* during $\mathcal{R}$A computation". Lack thereof allows self-relocating malware to move itself within $\mathcal{P}$rv memory during attestation, in order to avoid detection, e.g., if malware interrupts attestation execution, relocates itself to the part of *AR* that has already been covered by the integrity-ensuring function (*HMAC* in our case), and restarts attestation.

In higher-end devices, memory locking can be used to prevent modifications until the end of attestation, as discussed in [22]. However, in low-end devices, where applications run on bare-metal and there is no architectural support for memory locking, temporal consistency is attained by enforcing that attestation software (`SW-Att`) runs atomically: once it starts, it can not be interrupted by any software running on $\mathcal{P}$rv, thus preventing malware from interrupting $\mathcal{R}$A and relocating itself. While effective for security purposes, this requirement conflicts with real-time requirements if $\mathcal{P}$rv serves a safety-critical and time-sensitive function.

Some prior remediation techniques proposed to enable interrupts while maintaining temporal consistency, with high probability. SMARM [37] is one such approach. (Others similar techniques are discussed in [38]). SMARM divides attested memory (*AR*) into a set of blocks which are attested in a randomized order. Attestation of one block remains atomic. However, interrupts are allowed between attestation of two blocks. Assuming that malware can not guess the index of the next block to be attested, even if interrupts are allowed, malware only has a certain probability of avoiding detection. If the entire attestation procedure is repeated multiple times, this probability can be made arbitrarily small.

We note that, given the *RATA* optimization discussed in Section 8.1, attestation can be computed faster. In particular, since most Pseudo Random Function (PRF) implementations use block sizes of at least 32 bytes, the atomic attestation of one block in a SMARM-type strategy can not be faster than the attestation on *LMT* in *RATA* ($|LMT| = 32$ Bytes). In addition, attestation of *LMT* provides information about the content of *AR* in its entirety, with no probability of evasion. We believe this makes *RATA* more friendly to safety-critical operations than existing approaches.

In such settings, we envision that *AR* would be attested in its entirety at system boot time (**Case-2** in Section 8.1), while subsequent $\mathcal{R}$A would be computed on *LMT* only (**Case-1** in Section 8.1). We note that, if *AR* is eventually modified, $\mathcal{P}$rv would need to fall back to **Case-2** for the next $\mathcal{R}$A computation, which takes time to run atomically. However, after an unauthorized modification to $\mathcal{P}$rv memory, it is unclear why one would still want to offer real-time guarantees to compromised software.

8.3 Collective $\mathcal{R}$A Protocols and Device-to-Device Malware Relocation

Collective $\mathcal{R}$A protocols (CRA) (aka swarm attestation) [13, 39–44] are a set of techniques that attest a large number of devices that operate together as a part of a larger system. CRA schemes typically assume hybrid $\mathcal{R}$A architectures on individual devices and look into how to attest many devices efficiently. One security problem that is typically out of scope on single-device $\mathcal{R}$A and becomes relevant in CRA settings is caused by migratory malware. This is an analog of intra-device self-relocating malware (discussed in Section 8.2) that appears in collective settings. Specifically, instead of moving around inside the memory of the same device, it migrates from device to device to avoid detection.

To guarantee detection of migratory malware, CRA result must convince $\mathcal{V}$rf that all devices were in a safe state **within the same time window**, implying that malware had no destination device to migrate and avoid detection. Consequently, if a single-device attestation result conveys a safe state only at some point in between the execution of **Request** and **Verify** algorithms, it is nearly impossible (especially, in the presence of network delays) to conclude that migratory malware is not present in the swarm. Although this problem is discussed in the CRA literature existing approaches either place it outside their adversarial model [13, 39, 40, 43], or make a strong assumption about clock synchronization among all devices

in the swarm [26, 41, 42, 44], so that all devices can be scheduled to run **Attest** at the same time.

> Construction 3 (CRA-*RATA*). *Let $S = \{\mathcal{P}\text{rv}_1, \ldots, \mathcal{P}\text{rv}_n\}$ denote a swarm of n devices individually equipped with $RATA_B$ hybrid $\mathcal{R}$A facilities. Let LMT_i be the value of LMT in $\mathcal{P}\text{rv}_i$. Also,* **Verify**$(\mathcal{P}\text{rv}_i)$ *denotes the verification algorithm of Construction 2 for $\mathcal{P}\text{rv}_i$. Consider a protocol in which:*
>
> *(1) $\mathcal{V}$rf executes $RATA_B$ protocol, as defined in Construction 2 with each $\mathcal{P}\text{rv}_i$ in parallel. Let $t(Req_i)$ denote the time when $\mathcal{V}$rf issued the request to $\mathcal{P}\text{rv}_i$.*
>
> *(2) $\mathcal{V}$rf collects all responses and computes* **Verify**$(\mathcal{P}\text{rv}_i)$ *for all $\mathcal{P}\text{rv}_i \in S$. It then uses the values of LMT_i to learn "since when" $\mathcal{P}\text{rv}_i$ has been in a valid state. We denote this time as $t(LMT_i)$.*

We claim that, by addressing the TOCTOU problem in the single-device setting, $RATA_B$ can be utilized to construct the first CRA protocol secure against migratory malware without relying on synchronization of the entire swarm. To see why this is the case, consider Construction 3. In this construction, TOCTOU-Security on individual devices allows $\mathcal{V}$rf to conclude that each $\mathcal{P}$rv was in a valid state within a fixed time interval. Therefore, by checking the overlap in the valid interval of all $\mathcal{P}$rv-s, $\mathcal{V}$rf can learn the time window in which the entire swarm was safe as a whole, or detect migratory malware when such time window does not exist. Theorem 8.1 states the concrete guarantee offered by Construction 3.

> Theorem 8.1. *In Construction 3, if for all $\mathcal{P}\text{rv}_i \in S$,* **Verify**$(\mathcal{P}\text{rv}_i)$ *in step 2 succeeds for some $t(LMT_i)$, then it must be the case that entire S was in a valid state in the time window defined by the interval:*
>
> $$(\boldsymbol{max}[t(LMT_1), \ldots, t(LMT_n)]\,,\ \boldsymbol{min}[t(Req_1), \ldots, t(Req_n)]) \quad (18)$$
>
> *assuming equation 18 constitutes a valid interval.*
> **Note:** *(a, b) is a valid interval if $a < b$.*

Proof. *(Sketch)* It follows directly from the observations that:
– Given $\mathcal{R}$A-Security, for each $\mathcal{P}\text{rv}_i \in S$, a valid response can not be produced before the time when $\mathcal{P}\text{rv}_i$ receives $\mathcal{C}$hal, which is strictly greater than $t(Req_i)$.
– Given TOCTOU-Security, for each $\mathcal{P}\text{rv}_i \in S$ with **Verify**$(\mathcal{P}\text{rv}_i) = 1$, its memory could not have been changed between $t(LMT_i)$ and the first call to **Attest** after $t(Req_i)$. □

8.4 Runtime Attestation

Runtime attestation focuses on detection of runtime/data-memory attacks, providing authenticated information about software execution on $\mathcal{P}$rv. While it seems unrelated to detection of retrospective program memory modifications, we argue that *RATA* can also offer improvement to runtime attestation techniques.

Proofs of execution (PoX) for embedded systems were recently explored in [6] (APEX). A PoX proves that a given operation on $\mathcal{P}$rv was performed through the execution of the expected code and to verifies that outputs were indeed produced by this execution. Control Flow Attestation (CFA) introduced in [3] (C-FLAT) allows $\mathcal{V}$rf to also verify whether software that executed on $\mathcal{P}$rv took a specific (or a set of) valid control path(s), thus enabling detection of code-reuse attacks.

We note that regular (or static) $\mathcal{R}$A is a common stepping stone in these functionalities. In C-FLAT, OAT [45], and Tiny-CFA [4], the executable must be instrumented with specific instructions to enable CFA and $\mathcal{R}$A is used to verify that such instructions were not removed or modified. Besides, even executions with the same control-flow may differ in terms of behavior or outputs if their instructions differ. Similarly, in APEX, a proof of execution to $\mathcal{V}$rf is obtained via attestation of execution metadata. However, without attesting the corresponding executable (in program memory), such a proof would have no meaning other than: "some code executed successfully".

In many applications, the same executable is expected to remain in memory for long periods of time, while its proper execution (or control-flow) must be verified repeatedly, per safety-critical embedded operation [45]. *RATA* optimization discussed in 8.1 can minimize the overhead of such successive runtime attestations.

To illustrate this concept we combined *RATA* with APEX and Tiny-CFA, which is implemented atop APEX. In APEX, all runtime overhead *vis-a-vis* cost of executing the same software without proving its successful execution to $\mathcal{V}$rf is caused by the cost of static $\mathcal{R}$A. Since APEX is implemented atop *VRASED*, we implemented a *RATA*-compliant version of APEX without changing neither the internal behavior of *RATA* hardware modules nor APEX hardware module itself. As such, this approach substantially reduced PoX and CFA computational costs (these savings are consistent with Figure 8), while requiring the same additional hardware cost as reported in Table 2.

9 RELATED WORK

– **Remote Attestation ($\mathcal{R}$A):** $\mathcal{R}$A techniques generally fall into three categories: hardware-based, software-based and hybrid. Hardware-based techniques [11, 21, 46, 47] either perform $\mathcal{R}$A using a dedicated autonomous hardware component (e.g., a TPM [11]), or require substantial changes to the underlying instruction set architecture in order to support execution of trusted software (e.g., SGX [48]). Such changes are too expensive for cost-sensitive low-end embedded devices. On the other end of the spectrum, software-based techniques [49–51] require no hardware security features; they perform $\mathcal{R}$A using a custom checksum function implemented entirely in software. Security of software-based techniques relies on a precise measurement timing, which is only applicable to settings where the communication delay between $\mathcal{V}$rf and $\mathcal{P}$rv is negligible and/or constant, e.g., communication between peripherals and a host CPU. Thus, software-based $\mathcal{R}$A is unsuitable for environments where $\mathcal{R}$A must be performed over the internet. Whereas, hybrid $\mathcal{R}$A is particularly suitable for low-end embedded devices. It provides the same security guarantees as hardware-based $\mathcal{R}$A, while minimizing modifications to underlying MCU hardware. Current hybrid $\mathcal{R}$A techniques [7–10, 14, 52] implement the integrity-ensuring function (e.g., MAC) in software, and use trusted hardware to control execution of this software, preventing any violations that might cause $\mathcal{R}$A security problems, e.g., gadget-based attacks [53] or key leakage. This paper represents a paradigm shift of hybrid $\mathcal{R}$A, by having trusted hardware additionally provide some context about $\mathcal{P}$rv memory state.
– **Temporal Aspects of $\mathcal{R}$A:** Besides TOCTOU, two other temporal aspects are essential for $\mathcal{R}$A security: First, temporal consistency [22] means guaranteeing that the $\mathcal{R}$A result reflects an instantaneous snapshot of $\mathcal{P}$rv attested memory at some point in time during $\mathcal{R}$A. Lack thereof allows self-relocating malware to escape detection by copying and/or erasing itself during $\mathcal{R}$A. Temporal

consistency is achieved by enforcing atomic (uninterruptible) execution of attestation code, or by locking attested memory (i.e., making it unmodifiable) during $\mathcal{R}$A execution. Second, when $\mathcal{R}$A is used on safety-critical and/or real-time devices [38], atomicity requirement might interfere with the real-time nature of $\mathcal{P}$rv application. To address this issues, SMARM [37] relaxes this requirement by using probabilistic malware detection. Meanwhile, ERASMUS [12] and SeED [26] are based on $\mathcal{P}$rv self-measurements, in order to detect transient malware that infects $\mathcal{P}$rv and leaves before the next $\mathcal{R}$A instance. See Section 4.3 for further discussion on these types of techniques. Atrium [35] deals with physical-hardware adversaries that intercept instructions as they are fetched to the CPU during attestation. Atrium refers to that issue as TOCTOU. Despite nomenclature, that issue is clearly different from *RATA* goal.

– **Formal Verification and $\mathcal{R}$A:** Formal verification provides significantly higher level of assurance, yielding provable security for protocol specifications and implementations thereof. Recently, several efforts focused on formal verification of security-critical services and systems [23, 54–58]. *VRASED* [10] realized a formally verified $\mathcal{R}$A architecture targeting low-end devices. Other formally verified security services were obtained by extending *VRASED* to derive remote proofs of software update, memory erasure and system-wide MCU reset [1]. APEX [6] builds on top of *VRASED* to develop a verified architecture for proofs of remote software execution on low-end devices [6]. RATA also builds on top of *VRASED*, extending it to provide TOCTOU security while retaining original verified guarantees. Relying on *VRASED* allows us to reason about *RATA* design and to formally verify its security properties. Nonetheless, *RATA* main concepts are applicable to other hybrid (and possibly hardware-based, such as [20]) $\mathcal{R}$A architectures.

10 CONCLUSIONS

In this paper, we design, prove security of, and formally verify two designs ($RATA_A$ and $RATA_B$) to secure $\mathcal{R}$A against TOCTOU-related attacks, which perform illegal binary modifications on a low-end embedded system, in between successive $\mathcal{R}$A instances. $RATA_A$ and $RATA_B$ modules are formally specified and verified using a model-checker. They are also composed with *VRASED* – a verified $\mathcal{R}$A architecture. We show that this composition is TOCTOU-secure using a reduction-based cryptographic proof. Our evaluation demonstrates that a TOCTOU-Secure design is affordable even for cost-sensitive low-end embedded devices. Also, i n most cases, it reduces $\mathcal{R}$A time complexity from linear to constant, in the size of the attested memory.

ACKNOWLEDGMENTS

This work was supported by funding from: the Semiconductor Research Corporation (SRC) Contract 2019-TS-2907, NSF Awards SATC-1956393 and CICI-1840197, a subcontract from Peraton (formerly Perspecta) Labs, as well as the Coordinating Center for Thai Government Science and Technology Scholarship Students (CSTS), National Science and Technology Development Agency (NSTDA).

REFERENCES

[1] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," in *ICCAD*, 2019.

[2] M. Ammar and B. Crispo, "Verify&revive: Secure detection and recovery of compromised low-end embedded devices," in *Annual Computer Security Applications Conference*, pp. 717–732, 2020.

[3] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *ACM CCS*, pp. 743–754, ACM, 2016.

[4] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 641–646, IEEE, 2021.

[5] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[6] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th USENIX Security Symposium (USENIX Security 20)*, (Boston, MA), USENIX Association, Aug. 2020.

[7] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, 2012.

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

[9] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)," in *Wisec*, 2017.

[10] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *USENIX Security*, 2019.

[11] Trusted Computing Group., "Trusted platform module (tpm)," 2017.

[12] X. Carpent, N. Rattanavipanon, and G. Tsudik, "ERASMUS: Efficient remote attestation via self-measurement for unattended settings," in *DATE*, 2018.

[13] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik, "Lightweight swarm attestation: a tale of two lisa-s," in *ASIACCS*, 2017.

[14] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *DATE*, 2014.

[15] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "Toctou, traps, and trusted computing," in *International Conference on Trusted Computing*, Springer, 2008.

[16] R. V. Steiner and E. Lupu, "Attestation in wireless sensor networks: A survey," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, p. 51, 2016.

[17] M. Geden and K. Rasmussen, "Hardware-assisted remote runtime attestation for critical embedded systems," in *2019 17th International Conference on Privacy, Security and Trust (PST)*, pp. 1–10, IEEE, 2019.

[18] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "RATA source code." https://github.com/sprout-uci/RATA, 2021.

[19] O. Girard, "openMSP430." https://opencores.org/projects/openmsp430, 2009.

[20] J. Noorman, J. V. Bulck, J. T. Mühlberg, *et al.*, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Trans. Priv. Secur.*, vol. 20, no. 3, 2017.

[21] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *USENIX Security Symposium*, pp. 479–494, USENIX Association, 2013.

[22] X. Carpent, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Temporal consistency of integrity-ensuring computations and applications to embedded systems security," in *ASIACCS*, 2018.

[23] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *CCS*, 2017.

[24] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV*, 2002.

[25] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *VLSI Design*, 2004.

[26] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "SeED: secure non-interactive attestation for embedded devices," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[27] F. M. Anwar and M. Srivastava, "Applications and challenges in securing time," in *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, 2019.

[28] R. Annessi, J. Fabini, and T. Zseby, "It's about time: Securing broadcast time synchronization with data origin authentication," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–11, IEEE, 2017.

[29] L. Narula and T. E. Humphreys, "Requirements for secure clock synchronization," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 4, pp. 749–762, 2018.

[30] X. Du and H.-H. Chen, "Security in wireless sensor networks," *IEEE Wireless Communications*, vol. 15, no. 4, pp. 60–66, 2008.

[31] S. Ganeriwal, S. Čapkun, C.-C. Han, and M. B. Srivastava, "Secure time synchronization service for sensor networks," in *Proceedings of the 4th ACM workshop on Wireless security*, pp. 97–106, 2005.

[32] Y. Lindell and J. Katz, *Introduction to modern cryptography*, ch. 4.3, pp. 109–113. Chapman and Hall/CRC, 2014.

[33] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2SMV: A tool for word-level verification," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, 2016.

[34] T. Instruments, "Msp430 ultra-low-power sensing & measurement mcus." http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html.

[35] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *Proceedings of the 36th International Conference on Computer-Aided Design*, pp. 384–391, IEEE Press, 2017.

[36] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead control flow attestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 24, ACM, 2017.

[37] X. Carpent, N. Rattanavipanon, and G. Tsudik, "Remote attestation of iot devices via SMARM: Shuffled measurements against roving malware," in *HOST*, 2018.

[38] X. Carpent, K. Eldefrawy, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Reconciling remote attestation and safety-critical operation on simple iot devices," in *DAC*, 2018.

[39] N. Asokan, F. Brasser, A. Ibrahim, A. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "SEDA: scalable embedded device attestation," in *ACM CCS*, pp. 964–975, ACM, 2015.

[40] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A. Sadeghi, and M. Schunter, "SANA: secure and scalable aggregate network attestation," in *ACM CCS*, pp. 731–742, ACM, 2016.

[41] A. Ibrahim, A. Sadeghi, G. Tsudik, and S. Zeitouni, "DARPA: device attestation resilient to physical attacks," in *WISEC*, pp. 171–182, ACM, 2016.

[42] F. Kohnhäuser, N. Büscher, S. Gabmeyer, and S. Katzenbeisser, "Scapi: a scalable attestation protocol to detect software and physical attacks," in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 75–86, ACM, 2017.

[43] F. Kohnhäuser, N. Büscher, and S. Katzenbeisser, "Salad: Secure and lightweight attestation of highly dynamic and disruptive networks," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pp. 329–342, ACM, 2018.

[44] I. D. O. Nunes, G. Dessouky, A. Ibrahim, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Towards systematic design of collective remote attestation protocols," in *ICDCS*, 2019.

[45] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1433–1449, IEEE, 2020.

[46] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot — A coprocessor-based kernel runtime integrity monitor," in *USENIX Security Symposium*, 2004.

[47] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *IEEE Symposium on Security and Privacy, SP 2012*, pp. 239–253, IEEE Computer Society, 2012.

[48] Intel, "Intel Software Guard Extensions (Intel SGX)." https://software.intel.com/en-us/sgx.

[49] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *USENIX Security Symposium*, 2003.

[50] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE Symposium on Research in Security and Privacy (S&P)*, (Oakland, California, USA), pp. 272–282, IEEE, 2004.

[51] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," *ACM SIGOPS Operating Systems Review*, December 2005.

[52] F. Brasser, B. E. Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: tiny trust anchor for tiny devices," in *DAC*, pp. 34:1–34:6, ACM, 2015.

[53] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS '07*, 2007.

[54] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification.," in *OSDI*, 2014.

[55] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel, "Verified correctness and security of OpenSSL HMAC," in *USENIX*, 2015.

[56] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *IEEE S&P*, 2013.

[57] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, 2009.

[58] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "sel4: Formal verification of an os kernel," in *SIGOPS*, ACM, 2009.

[59] F. Brasser, K. B. Rasmussen, A. Sadeghi, and G. Tsudik, "Remote attestation for low-end embedded devices: the prover's perspective," in *DAC*, ACM, 2016.

[60] H. Krawczyk and P. Eronen, "HMAC-based extract-and-expand key derivation function (HKDF)," Internet Request for Comment RFC 5869, Internet Engineering Task Force, May 2010.

APPENDIX

A $\mathcal{V}$RF AUTHENTICATION DETAILS

```
1  void Hacl_HMAC_SHA2_256_hmac_entry() {
2      uint8_t key[64] = {0};
3      uint8_t verification[32] = {0};
4      if (memcmp(CHALL_ADDR, CTR_ADDR, 32) > 0)
5      {
6          memcpy(key, KEY_ADDR, 64);
7
8          hacl_hmac((uint8_t*) verification, (uint8_t*) key,
9                    (uint32_t) 64, *((uint8_t*)CHALL_ADDR),
10                   (uint32_t) 32);
11
12         if (!memcmp(VRF_AUTH, verification, 32)
13         {
14             hacl_hmac((uint8_t*) key, (uint8_t*) key,
15                 (uint32_t) 64, (uint8_t*) verification,
16                 (uint32_t) 32);
17             hacl_hmac((uint8_t*) MAC_ADDR, (uint8_t*) key,
18                 (uint32_t) 32, (uint8_t*) ATTEST_DATA_ADDR,
19                 (uint32_t) ATTEST_SIZE);
20             memcpy(CTR_ADDR, CHALL_ADDR, 32);
21         }
22     }
23
24     return();
25 }
```

Figure 9: `SW-Att` Implementation with $\mathcal{V}$rf authentication [10].

To prevent an adversary from impersonating $\mathcal{V}$rf and sending fake attestation requests to $\mathcal{P}$rv, *VRASED* design supports authentication of $\mathcal{V}$rf as part of `SW-Att` execution. The implementation is based on the protocol in [59]. In this protocol, $Chal$ is chosen by $\mathcal{V}$rf as a monotonically increasing nonce, i.e., consecutive requests i and $i+1$, $Chal_i < Chal_{i+1}$.

Figure 9 shows *VRASED* C implementation of `SW-Att`, including $\mathcal{V}$rf authentication. It also builds upon HACL* verified HMAC to authenticate $\mathcal{V}$rf, in addition to computing the authenticated integrity check over *AR*. In this case, $\mathcal{V}$rf request also contains an HMAC of the challenge computed using $\mathcal{K}$. Before calling `SW-Att`, software running on $\mathcal{P}$rv is expected to store the received challenge on a fixed address *CHALL_ADDR* and the corresponding received HMAC on *VRF_AUTH*. `SW-Att` discards the attestation request if (1) the received challenge is less than or equal to the latest challenge, or (2) HMAC of the received challenge is mismatched. After that, it derives a new unique key using HKDF [60] from $\mathcal{K}$ and the received HMAC and uses it as the attestation key.

To support authentication, *VRASED* extends `HW-Mod` with two additional properties to make the memory region that stores $\mathcal{P}$rv counter immutable to untrusted applications, i.e., any software except `SW-Att`. Notably, the counter requires persistent and writable storage, because `SW-Att` needs to modify it at the end of each attestation execution.

B PROOF OF THEOREM 5.1

Proof. By contradiction, assume a polynomial $\mathcal{A}$dv that wins the game in Definition 4.1 with probability $Pr[\mathcal{A}\text{dv}, \mathcal{R}\text{A-TOCTOU-game}] > \text{negl}(l)$. Therefore, $\mathcal{A}$dv can

produce $t_{LMT}||\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ such that:

$$\textbf{Verify}^{\mathcal{V}\mathrm{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathit{Chal}, M, t_0, t_{LMT}) = 1$$
$$\text{and}$$
$$\exists_{t_0 \le t_i \le t_{att}}\{AR(t_i) \neq M\}$$

By definition, **Verify** in Construction 1 results in 1 only if $t_{LMT} < t_0$. If $\mathcal{A}\mathsf{dv}$ simply replies with the actual value $t_{LMT} = LMT \geq t_i$, **Verify** result would be 0, since $t_i \geq t_0$, failing to satisfy **Verify** condition: $t_{LMT} < t_0$. Thus, to obtain **Verify** = 1, $\mathcal{A}\mathsf{dv}$ must spoof the value of t_{LMT} to $t_{LMT} < t_0$.

Upon receiving the spoofed value of t_{LMT} the **Verify** now expects:

$$\mathsf{H}_{\mathcal{A}\mathsf{dv}} \equiv HMAC(KDF(\mathcal{K}, MR), M) \tag{19}$$

where expected M reflects $LMT = t_{LMT}$, i.e., $LMT < t_0$.

Also, hardware enforced properties 11 and 12 guarantee that $LMT \in AR$ always contains the time of the most recent modification of AR. Thus, because $t_{att} \geq t_i$, it must be the case that $AR(t_{att})$ reflects $LMT \geq t_i$ implying $LMT \neq t_{LMT}$, and consequently $AR(t_{att}) \neq M$.

Under such restriction, $\mathcal{A}\mathsf{dv}$ ability to win the game implies its capability to produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ such that $\textbf{Verify}^{\mathcal{V}\mathrm{rf}}(\mathsf{H}, \mathit{Chal}, M, t_0, t_{LMT}) = 1$, even though modifying AR such that $AR(t_{att}) = M$ is not possible. To conclude the proof, we show that the existence of such an $\mathcal{A}\mathsf{dv}$ implies the existence of another adversary $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ that wins the $\mathcal{R}\mathsf{A}$ security game in Definition 3.2 against *VRASED*, contradicting the theorem's assumption.

To win the game in Definition 3.2 $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ behaves as follows:

(1) At time t_i where $t_0 \leq t_i \leq t_{att}$, $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ modifies AR causing $LMT \in AR$ to store the value of t_i.
(2) $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ receives Chal from the challenger in step (2) of $\mathcal{R}\mathsf{A}$ security game of Definition 3.2 and executes the same algorithm of $\mathcal{A}\mathsf{dv}$ with inputs Chal and $t_{att} = t$ to produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$, such that $\textbf{Verify}^{\mathcal{V}\mathrm{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathit{Chal}, M, t_0, t_{LMT}) = 1$ with probability:
$$Pr[\mathcal{A}\mathsf{dv}, \mathcal{R}\mathsf{A}\text{-TOCTOU-game}] > \mathsf{negl}(\mathsf{l}),$$
even though $t_{LMT} < t_0 < t_i$.
(3) As a response in step 3 of the game in Definition 3.2, $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ replies with: $\sigma = \mathsf{H}_{\mathcal{A}\mathsf{dv}}$.

Since $\textbf{Verify}^{\mathcal{V}\mathrm{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathit{Chal}, M, t_0, t_{LMT}) = 1$, it follows that $\sigma = \mathsf{H}_{\mathcal{A}\mathsf{dv}} = HMAC(KDF(\mathcal{K}, MR), M)$, for expected M containing $LMT = t_{LMT}$. However, due to the AR modification at time t_i, $AR(t)$ must reflect $LMT \geq t_i$, satisfying the condition that $AR(t) \neq M$ and allowing $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ to win the game in Definition 3.2 with probability:

$$Pr[\mathcal{A}\mathsf{dv}, \mathcal{R}\mathsf{A}\text{-game}] = \mathsf{Pr}[\mathcal{A}\mathsf{dv}, \mathcal{R}\mathsf{A}\text{-TOCTOU-game}] > \mathsf{negl}(\mathsf{l}) \tag{20}$$

□

C PROOF OF THEOREM 6.1

We now show that, if properties in Equations 15, 16 and 17 hold, existence of $\mathcal{A}\mathsf{dv}$ that wins the TOCTOU security game against $RATA_B$ implies the existence of another $\mathcal{A}\mathsf{dv}$ that wins $\mathcal{R}\mathsf{A}$ security game against *VRASED*, thus contradicting the initial premise.

Proof. By contradiction, assume a polynomial $\mathcal{A}\mathsf{dv}$ that wins the game in Definition 4.1 with probability $Pr[\mathcal{A}\mathsf{dv}, \mathcal{R}\mathsf{A}\text{-TOCTOU-game}] > \mathsf{negl}(\mathsf{l})$. Therefore, $\mathcal{A}\mathsf{dv}$ can produce response $LMT_{\mathcal{A}\mathsf{dv}}||\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ such that:

$$\textbf{Verify}^{\mathcal{V}\mathrm{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathit{Chal}, M, t_0, T, LMT_{\mathcal{A}\mathsf{dv}}) = 1$$
$$\text{and}$$
$$\exists_{t_0 \le t_i \le t_{att}}\{AR(t_i) \neq M\}$$

By definition, in Construction 2, **Verify** outputs 0 if $LMT_{\mathcal{A}\mathsf{dv}}$ differs from Chal_P stored by $\mathcal{V}\mathrm{rf}$ in the challenge-time association pair $P = (\mathit{Chal}_P, t_P)$. If $LMT_{\mathcal{A}\mathsf{dv}} = \mathit{Chal}_P$, it corresponds to a challenge value sent before t_0 (assuming sensible choices of t_0 by $\mathcal{V}\mathrm{rf}$). Therefore, in order to win, $\mathcal{A}\mathsf{dv}$ **must choose** $LMT_{\mathcal{A}\mathsf{dv}} = \mathit{Chal}_P$.

Since $LMT \in AR$, by claiming a value for $LMT_{\mathcal{A}\mathsf{dv}}$ fitting the restriction above, $\mathcal{A}\mathsf{dv}$ causes the expected memory value M to also reflect, $LMT = LMT_{\mathcal{A}\mathsf{dv}}$. At this point, $\mathcal{A}\mathsf{dv}$ has two possible actions: to modify AR to call **Attest** with $AR(t_{att}) = M$; or to obtain $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ even with $AR(t_{att}) \neq M$. First we show that the latter is $\mathcal{A}\mathsf{dv}$'s only option.

Suppose that $\mathcal{A}\mathsf{dv}$ attempts to set $AR(t_{att}) = M$ to call **Attest**. In this case, we stress three observations about $RATA_B$:

(1) By LTL statement 17, any modification to AR in between the i-th and $(i+1)$-th authenticated computations of **Attest**, will cause AR to change to reflect $LMT = \mathit{Chal}_{i+1}$ in following $\mathcal{R}\mathsf{A}$ responses. Therefore, the premise that:
$$\exists_{t_0 \le t_i \le t_{att}}\{AR(t_i) \neq M\}$$
will necessarily update LMT.
(2) From *VRASED* authentication (see Appendix A), for subsequent $\mathcal{R}\mathsf{A}$ challenges Chal_i and Chal_{i+1} that authenticate successfully, it is always the case that $\mathit{Chal}_i < \mathit{Chal}_{i+1}$.
(3) From LTL statement 16, $RATA_B$ never updates LMT with a challenge if it does not authenticate successfully. Since authentication implies $\mathit{Chal}_i < \mathit{Chal}_{i+1}$, a call to **Attest** never causes LMT to be updated to a previously used Chal.

From observations 1, 2, and 3 above, it is impossible to set $AR = M$ by calling **Attest**, because any modification to LMT caused by **Attest** will always change LMT to a value that was never used before and thus different from Chal_P. At this point $\mathcal{A}\mathsf{dv}$ last resource is to try to write to LMT directly. However, this is immediately in conflict with LTL property 15. Since making $AR(t_{att}) = M$ is impossible after a modification at time t_i, the assumption that $\mathcal{A}\mathsf{dv}$ wins the game in Definition 4.1 implies that $\mathcal{A}\mathsf{dv}$ can produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ that verifies successfully even when $AR(t_{att}) \neq M$. To conclude the proof, we show that existence of such $\mathcal{A}\mathsf{dv}$ implies existence of another adversary $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ that wins the $\mathcal{R}\mathsf{A}$ security game in Definition 3.2.

To win the game in Definition 3.2 $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ is constructed as follows:

(1) At time some t_i, where $t_0 \leq t_i \leq t$, $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ modifies memory in AR.
(2) $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ receives Chal in step 2 of $\mathcal{R}\mathsf{A}$ security game of Definition 3.2, and executes the same algorithm as $\mathcal{A}\mathsf{dv}$ on Chal and with $t_{att} = t$ to produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$, such that $\textbf{Verify}^{\mathcal{V}\mathrm{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathit{Chal}, M, t_0, T, LMT_{\mathcal{A}\mathsf{dv}}) = 1$ with probability:

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

(3) As a response in step 3 of the game in Definition 3.2, $\mathcal{A}\text{dv}_{\mathcal{R}\text{A}}$ replies with $\sigma = \text{H}_{\mathcal{A}\text{dv}}$.

Since $\textbf{Verify}^{\mathcal{V}\text{rf}}(\text{H}_{\mathcal{A}\text{dv}}, \mathcal{C}\text{hal}, M, t_0, T, LMT_{\mathcal{A}\text{dv}}) = 1$, it follows that $\sigma = HMAC(KDF(\mathcal{K}, \mathcal{C}\text{hal}), M)$ (first condition for $\mathcal{A}\text{dv}_{\mathcal{R}\text{A}}$ to win), for expected M containing $LMT = LMT_{\mathcal{A}\text{dv}}$. On the other hand, because memory was modified at time t_i, it must be the case that $AR(t)$ has $LMT \neq LMT_{\mathcal{A}\text{dv}}$. Thus satisfying the remaining condition that $AR(t) \neq M$ implies that $\mathcal{A}\text{dv}_{\mathcal{R}\text{A}}$ wins the game in Definition 3.2 with probability:

$$Pr[\mathcal{A}\text{dv}, \mathcal{R}\text{A-game}] = \text{Pr}[\mathcal{A}\text{dv}, \mathcal{R}\text{A-TOCTOU-game}] > \text{negl}(\text{l}) \tag{21}$$

□

D *RATA* IMPLEMENTATION WITH SANCUS

To demonstrate *RATA* generality, we also implemented it atop SANCUS [20]: a hardware-based $\mathcal{R}$A architecture targeting the same class of embedded devices. To the best of our knowledge, aside from VRASED (used in our verified implementation), SANCUS is the only other open-source $\mathcal{R}$A architecture for low-end embedded systems, which justifies our choice. We note that this implementation is intended to demonstrate *RATA* generality and that provable security guarantees derived from *RATA*-with-*VRASED* do not apply here. Since SANCUS does not provide a formal security model and analysis, provable composition of RATA atop SANCUS is not currently possible.

Since *RATA* operates as a standalone monitor that does not interfere with neither the CPU nor the underlying $\mathcal{R}$A architecture functionality, adapting RATA to work with SANCUS is almost effortless. We describe this implementation in terms of $RATA_A$, which is simpler and does not depend on $\mathcal{V}$rf authentication. The main difference from the *VRASED*-based implementation is due to SANCUS support for isolated software modules (SMs), where each SM is attested individually as an independent program. We note that even SANCUS' support for attestation and inter-process isolation is insufficient to provide TOCTOU-Security, since $\mathcal{P}$rv program memory could be physically re-programmed or modified via exploits to vulnerabilities in the code of the isolated application itself, without $\mathcal{V}$rf's knowledge. Hence, similar to *VRASED*'s $\mathcal{R}$A case, *RATA* also complements SANCUS security guarantees.

To enable *RATA* functionality over SANCUS one must be careful (when programming $\mathcal{P}$rv) to configure the software binary such that the program memory of a particular SM of interest coincides with *RATA AR* region. As such, program memory of the SM will be automatically checked by *RATA* module and SANCUS attestation of such SM program memory will also cover *LMT* (since $LMT \in AR$) providing an authenticated proof to $\mathcal{V}$rf of the time of the latest modification of such SM program memory.

We note that this approach requires one *RATA* module per SM, since multiple SMs imply dividing $\mathcal{P}$rv program memory into multiple *ARs* and corresponding LMT regions. Nonetheless, since low-end devices typically run very few processes, we expect the cost to remain manageable.

Because SANCUS is implemented on the same MCU as *VRASED* (OpenMSP430), no internal modifications are required to *RATA* hardware module, and its additional hardware cost remains consistent with that reported in Table 2. To support TOCTOU-Secure attestation of multiple SMs, this cost grows linearly, i.e., the cost incurred by one *RATA* hardware module multiplied by the number of independent SMs that should support TOCTOU-Secure attestation. We note that, in $RATA_A$, the same secure read-only synchronized clock can be shared by all such modules.