Invited: Reconciling Remote Attestation and Safety-Critical Operation on Simple IoT Devices

Xavier Carpent
UC Irvine
xcarpent@uci.edu

Karim Eldefrawy
SRI International
karim.eldefrawy@sri.com

Norrathep Rattanavipanon
UC Irvine
nrattana@uci.edu

Ahmad-Reza Sadeghi
TU Darmstadt
ahmad.sadeghi@trust.cased.de

Gene Tsudik
UC Irvine
gene.tsudik@uci.edu

ABSTRACT

Remote attestation (RA) is a means of malware detection, typically realized as an interaction between a trusted verifier and a potentially compromised remote device (prover). RA is especially relevant for low-end embedded devices that are incapable of protecting themselves against malware infection. Most current RA techniques require on-demand and uninterruptible (atomic) operation. The former fails to detect transient malware that enters and leaves between successive RA instances; the latter involves performing potentially time-consuming computation over prover's memory and/or storage, which can be harmful to the device's safety-critical functionality and general availability. However, relaxing either on-demand or atomic RA operation is tricky and prone to vulnerabilities. This paper identifies some issues that arise in reconciling requirements of safety-critical operation with those of secure remote attestation, including detection of transient and self-relocating malware. It also investigates mitigation techniques, including periodic self-measurements as well as interruptible attestation modality that involves shuffled memory traversals and various memory locking mechanisms.

ACM Reference Format:
Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. 2018. Invited: Reconciling Remote Attestation and Safety-Critical Operation on Simple IoT Devices. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3199853

1 INTRODUCTION

Prior to about 10 years ago, preferred targets of malware were general-purpose computers and, later, smartphones. However, in recent years, the number and variety of special-purpose computing devices has increased dramatically. This includes all kinds of embedded devices, cyber-physical systems (CPS) and Internet-of-Things (IoT) gadgets. They are increasingly occurring in various "smart" settings, such as home, office, factory, automotive and public venues. Despite many benefits, these devices unfortunately also represent natural and attractive malware attack targets. This is mainly because they are numerous, inter-connected and/or connected to the Internet, and their security is either poor or non-existent.

As society becomes increasingly accustomed to being surrounded by, and deriving benefits from, such devices, their well-being becomes a paramount concern. In the context of actuation-capable devices, malware can impact security and safety, e.g., as demonstrated by Stuxnet [28]. Whereas, for sensing devices, malware can undermine privacy by obtaining ambient information. Furthermore, clever malware can turn vulnerable IoT devices into zombies that can become sources for DDoS attacks. For example, in Fall 2016, a multitude of compromised "smart" cameras and DVRs formed the Mirai Botnet [1] which was used to mount a massive-scale DDoS attack.

Security is typically not the highest priority for low-end device manufacturers, due to cost, size or power constraints, as well as the rush-to-market syndrome. It is thus unrealistic to expect such devices to have the means to prevent malware attacks. The next best thing is detection of malware presence. This typically requires some form of **Remote Attestation (RA)** – a distinct security service for detecting malware on CPS, embedded and IoT devices. RA is especially applicable to low-end embedded devices incapable of defending themselves against malware infection. This is in contrast to more powerful devices (both embedded and general-purpose) that can avail themselves of sophisticated anti-malware protection. RA involves verification of current internal state (i.e., RAM and/or flash) of an untrusted remote hardware platform (prover or $\mathcal{P}$rv) by a trusted entity (verifier or $\mathcal{V}$rf). If $\mathcal{V}$rf detects malware presence, $\mathcal{P}$rv's software can be re-set or rolled back and out-of-band measures can be taken to prevent similar infections. In general, RA can help $\mathcal{V}$rf establish a static or dynamic root of trust in $\mathcal{P}$rv and can also be used to construct other security services, such as software updates [25] and secure deletion [21].

2 BACKGROUND AND MOTIVATION

2.1 RA Overview

Many RA techniques with various assumptions, security features and complexities have been proposed. Most of them can be divided into three approaches: *hardware-based, software-based, and hybrid.*

Hardware-based approaches typically rely on security provided by a separate and dedicated secure hardware component, such as a

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3199853

Trusted Platform Module (TPM) [27]. Despite resisting all, except physical attacks, hardware-based approaches are unsuitable for low-end and legacy embedded devices due to its added complexity and various cost factors.

Software-based RA techniques offer a very low-cost alternative. Pioneer [26] is a prominent example of this approach; it relies on a one-time special checksum function that covers memory in an unpredictable (rather than contiguous) fashion. Any interference with, or emulation of, the computation of this checksum is detectable by extra latency incurred by self-relocating malware moving itself (in parts) while trying to avoid being "caught" by the checksum. Unfortunately, security of this approach is uncertain after several attacks on software-based RA schemes (e.g., [8]) were demonstrated. Another problem with the software-based approach is that it requires strong assumptions about adversarial capabilities, which are unrealistic in many real settings. However, this is the only RA option for legacy devices.

Hybrid (software-hardware) RA co-designs attempt to overcome limitations of purely software-based techniques while minimizing hardware requirements. Hybrid RA is also especially suitable for mid-range and low-end embedded devices, which usually lack, or cannot accommodate, a secure hardware component, such as a TPM. SMART [12] is the first hybrid RA architecture with minimal hardware modifications to existing microcontroller units (MCUs). SMART requires uninterruptible and atomic execution of non-malleable ROM-resident attestation code which has exclusive access to attestation key(s); this is enforced by hard-wired MCU access control rules. Actual attestation is performed by $\mathcal{P}$rv computing a cryptographic checksum over a specific memory region and returning the result to $\mathcal{V}$rf. Notably, SMART's requirement for atomic execution of attestation code was motivated by the need to mitigate code-reuse, e.g., ROP [22] attacks.

HYDRA [10, 11] implements SMART for devices with a Memory Management Unit (MMU). It builds upon the formally verified seL4 [18] microkernel, which ensures process memory isolation and enforces access control to memory regions. Given guaranteed process isolation features of seL4, SMART access control rules are implemented in software and enforced by seL4 in the HYDRA security architecture. Similar to SMART, HYDRA requires execution of the attestation process to be atomic. HYDRA achieves this property by starting the attestation process with the highest priority, while assigning lower priorities to all other processes. It requires hardware-enforced secure boot to securely instantiate seL4 and the attestation process.

The TrustLite [19] security architecture also supports RA for low-end devices. It differs from SMART in two ways: First, interrupts are allowed and handled securely by the CPU Exception Engine. Second, static access control rules can be programmed in software using an Execution-Aware Memory Protection Unit (EA-MPU). A follow-on effort, called TyTAN [3], adopts a similar approach while providing additional real-time guarantees and dynamic configuration for safety- and security-critical applications.

Aforementioned RA techniques operate in a single prover setting. However, various emerging applications such as [24] require attesting a group (or a swarm) of interconnected embedded devices. In that setting, it is beneficial to take advantage of interconnectivity and perform collective attestation using a dedicated protocol. Several techniques [2, 4, 23] for efficient RA of large and dynamic device swarms have been proposed. Attestation results from these techniques in provide a range of information about the state of the attested devices. Another swarm RA technique, DARPA [13], additionally offers a means to detect physical attacks.

2.2 On-Demand RA Timeline

In general terms, the attestation process ($\mathcal{M}$P) on $\mathcal{P}$rv computes a keyed integrity-ensuring function. The input to this function (i.e., the aforementioned cryptographic checksum) is all, or part, of $\mathcal{P}$rv's memory. An on-demand RA scheme proceeds as follows:

(1) $\mathcal{V}$rf sends a challenge-bearing attestation request to $\mathcal{P}$rv
(2) $\mathcal{P}$rv receives it and starts $\mathcal{M}$P
(3) $\mathcal{P}$rv finishes $\mathcal{M}$P and sends the attestation report to $\mathcal{V}$rf
(4) $\mathcal{V}$rf receives the report from $\mathcal{P}$rv and verifies it

The timeline illustrating this sequence of events is shown in Figure 1. Computation of $\mathcal{M}$P (in gray) starts at t_s and ends at t_e. However, in practice, it may be deferred on $\mathcal{P}$rv due to networking delays, $\mathcal{V}$rf's request authentication, or termination of the previously running task.

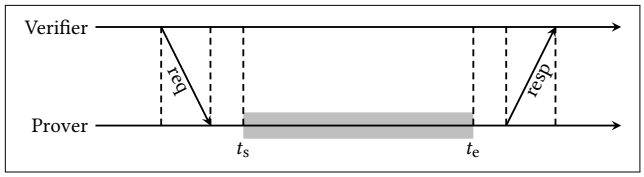

Figure 1: Timeline for an on-demand RA scheme.

2.3 RA Coverage

The usual RA coverage on $\mathcal{P}$rv includes executable code residing in RAM or in some non-volatile memory. RA might also cover non-executable regions on $\mathcal{P}$rv, i.e., data. Let M, of bit-size L, represent $\mathcal{P}$rv's memory to be attested. If content of M is known *a priori* to $\mathcal{V}$rf and expected to be immutable, then $\mathcal{P}$rv can execute $\mathcal{M}$P over M and send the result to $\mathcal{V}$rf, which can easily validate it. The same applies if M is mutable and its entropy is low: $\mathcal{V}$rf can compute (or pre-compute) all possible valid (benign) results of $\mathcal{M}$P over M and thus validate $\mathcal{P}$rv's result. However, if entropy of M is high, enumeration of its possible valid states can become infeasible. This is likely to occur when parts of M correspond to data, e.g., stack, heap and registers. One way to address this issue is for $\mathcal{P}$rv to return to $\mathcal{V}$rf the actual contents of parts of M that are highly mutable. For example, if $M = [C, D]$ where C represents immutable code and D – volatile high-entropy data region(s), $\mathcal{P}$rv can return the fixed-size measurement result produced by $\mathcal{M}$P over M, accompanied by a copy of D. Clearly, this only makes sense if $|D|$ is small, i.e., $|D| << L$. Furthermore, if content of D is irrelevant to $\mathcal{V}$rf, $\mathcal{P}$rv can easily zero it out before executing $\mathcal{M}$P. This makes it impossible for malware to hide in such regions, and obviates the need for $\mathcal{P}$rv to send $\mathcal{V}$rf an explicit copy of D.

2.4 Timing Overhead

Timing complexity (overhead) of $\mathcal{M}$P on $\mathcal{P}$rv is dominated by measurement of attested memory, which, in turn, depends on the size

of that memory, $\mathcal{P}$rv's computational capabilities, and underlying cryptographic function(s). One natural way to obtain a measurement with hybrid or hardware RA techniques[1] is by computing a Message Authentication Code (MAC), based either on hashing (e.g., HMAC-SHA-2 [20]) or encryption (e.g., AES-CBC-MAC [15]). We focus on hash-based MACs. Alternatively, a measurement can be obtained by computing a Digital Signature, via the standard hash-and-sign method, e.g., using RSA [17] or EC-DSA [16]. MACs are clearly much cheaper than signatures. Whereas, if non-repudiation or strong origin authentication is required, signatures are justified.

Regardless of MACs or signatures, timing overhead is mostly determined by hashing. The actual signature time is independent of memory size, since only the fixed-size hash is actually signed. Of course, for small memory sizes, signature computation is the main cost component. However, for any signature algorithm, there is a point at which the cost of hashing exceeds that of signing. Also, for HMAC-based MACs, the cost of the outer hash is negligible compared to the inner one which processes actual data.

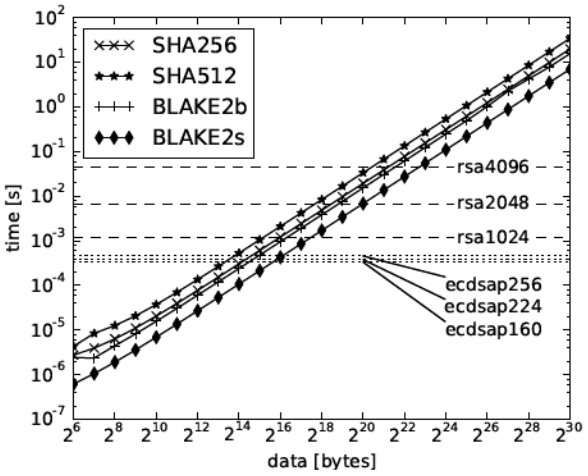

Figure 2: Timings of several hash functions and signatures on ODROID-XU4.

We now illustrate some concrete timing measurements. As a sample $\mathcal{P}$rv hardware platform, we use ODROID-XU4 [9], a popular single-board MCU representative of medium-to-low-end embedded systems. Figure 2 shows timings of $\mathcal{M}$P for various memory sizes, and for several hash and signature choices. We picked some popular hash functions: SHA-256, SHA-512, Blake2b and Blake2s (the latter two are in particular well suited for embedded systems), as well as popular signature schemes: RSA-1024, RSA-2048, RSA-4096, ECDSA-160, ECDSA-224, and ECDSA-256.

As illustrated in Figure 2, for input sizes over 1MB, $\mathcal{M}$P takes longer than 0.01sec, and the cost of most signature algorithms become comparatively insignificant. Results show that even hashing a reasonably small amount of memory incurs a significant delay, e.g., about 0.9sec to measure just 100MB on ODROID-XU4. Measuring its entire RAM (2GB) is quite time-consuming at nearly 14sec.

[1] As mentioned earlier, software-based RA techniques use custom checksums, rather than cryptographic functions.

2.5 RA in Safety-Critical Settings

Since low-end devices are often used in real-time and safety-critical applications, it is important to minimize the impact of RA on normal operation (i.e., availability) of such devices. In particular, it might be undesirable to allow attestation code on $\mathcal{P}$rv to run without interruption, considering that computing a measurement over a substantial amount of memory might take a relatively long time, as shown above in Section 2.4.

For example, consider a sensor-actuator fire alarm application running over "bare-metal" on a low-end embedded $\mathcal{P}$rv, powered by ODROID-XU4. This application periodically (say, every second) checks the value of its temperature sensor and triggers an alarm whenever that value exceeds a certain threshold. Given its safety-critical function, software integrity of $\mathcal{P}$rv is periodically validated via RA, where the role of $\mathcal{V}$rf is played by a fire-alarm controller or a smart control panel. Upon receipt of a request from $\mathcal{V}$rf, $\mathcal{M}$P interrupts the critical application and takes over. Using a hybrid RA technique (e.g., SMART), $\mathcal{M}$P must run uninterrupted in order to accurately reflect $\mathcal{P}$rv's current state. Assuming attested memory size of 1GB, $\mathcal{M}$P would run for approximately 7sec. However, if an actual fire breaks out soon after $\mathcal{M}$P starts, it would take a very long time for the application to regain control, sense the fire and sound the alarm. Precious time lost as a result of non-interruptible $\mathcal{M}$P might cause disastrous consequences.

At this point, it is natural to conclude that the atomicity requirement should be relaxed and $\mathcal{M}$P should be interruptible by a legitimate time-critical application. Unfortunately, allowing interrupts in the attestation code opens the door for malware detection evasion strategies. For example, if $\mathcal{P}$rv is compromised, its time-critical application might contain malware which presumably wants to avoid detection. When confronted with imminent attestation and thus subsequent detection, it may want to:

- Simply erase itself, perhaps in order to reappear later. This is an example of **transient malware**. More generally, transient malware is one that infects $\mathcal{P}$rv and later leaves, ideally leaving no trace.
- Remain on $\mathcal{P}$rv and try to avoid detection by moving itself around *during* attestation. This behavior corresponds to **self-relocating malware**.

Therefore, it becomes clear that there is an inherent conflict between the needs of safety-critical applications and RA security requirements. Resolving this conflict represents a major challenge that we explore in the remainder of this paper.

3 SOLUTION LANDSCAPE

In this section, we consider some potential approaches to reconcile security requirements of RA with those of safety-critical applications. Table 1 presents a summary of properties and specificities of these potential solutions.

3.1 Memory Locking

As mentioned earlier, hybrid RA architectures, such as TrustLite [19] and TyTAN [3], permit tasks to be interrupted. While this allows for time-critical processes to run and preserve $\mathcal{P}$rv's critical functionality, attestation results might be *inconsistent*. Indeed, in TrustLite,

Table 1: Summary of features provided by potential solutions.

Solutions		Malware Detection		Writable Mem. Availability	Consistency Guarantees	Interruptibility	Unattended Setting	Extra HW Requirements	Run-Time Overhead
		Self-relocating	Transient						
Baseline: SMART-based On-Demand RA [12]		✓	✓	✗	✓	✗	✗	Baseline	Baseline
Memory Locking [5]	All-Lock	✓	✓	✗	✓	✓ (to some degree)	✗	Dynamically Configurable MPU or MMU	Low
	Dec-Lock	✓	✓	✓ (to some degree)					
	Inc-Lock	✓	✗	✓ (to some degree)					
Shuffled Measurement [7]		✓ (high prob.)	✗	✓	✗	✓ (to some degree)	✗	None (optionally Secure Memory)	High
Self-Measurement [6, 14]		✓	✓	✗	✓	✗ (may be made context aware)	✓	Secure Clock	None

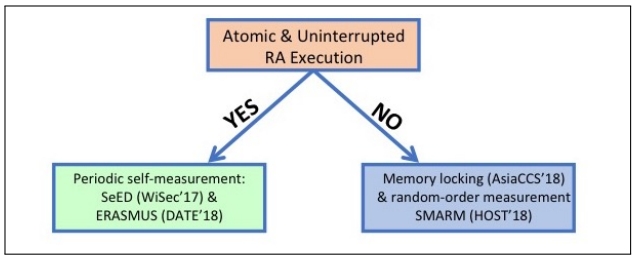

Figure 3: Overview of potential solutions.

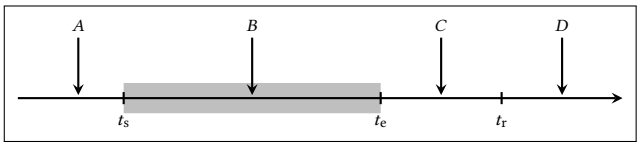

Figure 4: Timeline for computation of F. It starts at t_s and ends at t_e. Consistency is considered until t_r. A change to M at time A or D has no effect. Impact of a change at time B or C depends on the consistency mechanism.

since memory can change *during* execution of $\mathcal{M}$P, the report produced and sent to $\mathcal{V}$rf might correspond to a state of $\mathcal{P}$rv's memory that *never existed in its entirety* at any given time. This is problematic if $\mathcal{V}$rf is infected with self-relocating malware. Assuming that such malware resides in the second half of $\mathcal{P}$rv's memory, it can interrupt $\mathcal{M}$P after the latter covers the first half of $\mathcal{P}$rv's memory, copy itself into the first half, erase traces in its former location, and resume $\mathcal{M}$P. This way, malware remains undetected despite the fact that all memory locations have been measured.

In TyTAN [3], memory of each process is measured individually. While higher-priority processes may interrupt $\mathcal{M}$P to meet real-time requirements, the process being measured may not do so, regardless of its priority. This may protect against a single-process malware from moving itself in memory. However, malware that is spread over several colluding processes can defeat this countermeasure. Doing so would require malware to violate process isolation, e.g., by exploiting an OS vulnerability. Also, in a low-end device with a single task (besides $\mathcal{M}$P), this corresponds to uninterruptible operation.

SMART [12] disables interrupts as the first step in $\mathcal{M}$P. This precludes self-relocating malware. Uninterruptibility is required to protect the attestation key and to ensure that $\mathcal{M}$P is performed from beginning to end. However, temporal consistency was not an explicit design goal of SMART. Consequently, although it *coincidentally* guarantees consistency, SMART is unsuitable for time- or safety-critical applications.

Several mechanisms that offer various tradeoffs between consistency guarantees and real-time requirements were proposed and prototyped over the HYDRA hybrid-RA architecture in [5]. Consistency is achieved through *locking* memory regions, i.e., temporarily making them read-only. Locking can be realized via system-calls and capabilities enabled by a secure microkernel that is supported by underlying hardware features, e.g., as in seL4 [18] microkernel.

Three points in the timeline of computation of an integrity-ensuring function [2] F are relevant to our discussion (see Figure 4):

(1) t_s– computation of F *starts*;
(2) t_e– computation *ends*;
(3) Optionally, t_r– $\mathcal{P}$rv is explicitly requested to *release* a current memory lock.

3.1.1 Basic Approaches. There are three obvious options:

(1) `No-Lock`:The simplest mechanism is a strawman that does not lock memory. The result is computed using contents of each memory block at the time when F processes it, which means that it provides no consistency guarantees. Thus, it might not detect self-relocating or transient malware.
(2) `All-Lock`: The other extreme is to lock the entire memory M at t_s, and leave it locked throughout computation of F, finally releasing it all at t_e. This provides very strong temporal consistency guarantees at the cost of being very restrictive and unfriendly to interrupting (potentially critical) tasks that may require modifying locked memory. The measurement is consistent with M within $[t_s, t_e]$. This also implies that M is immutable and thus constant from t_s to t_e.
(3) `All-Lock-Ext`: An extended variant of `All-Lock` that provides extra consistency keeps all memory locked until t_r. Similar to `All-Lock`, the measurement remains consistent with M at every $[t_s, t_r]$, and M stays constant from t_s to t_r. An extended lock can be advantageous if $\mathcal{V}$rf wishes to guarantee that $\mathcal{P}$rv is in a given state at a particular time t_r, as opposed to "some time in the past".

3.1.2 Sliding-Lock Approaches. Several sliding-lock mechanisms that dynamically lock or unlock blocks of memory during execution of F were proposed in [5]. In the Decreasing Lock (`Dec-Lock`) mechanism, the entire M is locked at t_s, and each block is released as soon as F completes processing it. The measurement is consistent

[2] An integrity-ensuring function (e.g., MAC) F is a main component of $\mathcal{M}$P.

with all of M at time t_s only. This implies detection of any malware present in M at t_s. The Increasing Lock (`Inc-Lock`) mechanism locks blocks as they are processed. Entire M is unlocked at t_s and it is gradually locked as computation of F proceeds, until it is completely locked at t_e, after which it is fully released. In other words, each memory block is locked only when it is required for F. The measurement is consistent with M at t_e only. This implies detection of self-relocating, though not transient, malware. Unlike `Dec-Lock`, it is beneficial to end the computation of F with blocks that require high availability, since they are locked for the shortest time. As with `All-Lock-Ext`, it is possible to add extra-computation consistency to `Inc-Lock` by only releasing the lock at t_r, instead of t_e. Then, the measurement is consistent with M within the entire interval $[t_e, t_r]$, and M stays constant within $[t_e, t_r]$. This type of extension is not naturally applicable to `Dec-Lock` since memory is not locked at t_e. For more details on other mechanisms and extensions, along with implementation details and experimental results, we refer to [5].

3.2 Shuffled Measurements

Making $\mathcal{M}$P interruptible without memory locking (see `No-Lock` in Section 3.1.1) may be vulnerable to self-relocating malware. It is nevertheless the approach taken in SMARM [7], with the following additional twist: memory is measured in a random, secret order. This order is established randomly when the measurement starts, and then stored in secure memory (or encrypted in non-secure memory).

SMARM makes the realistic assumption that malware is unable to determine what blocks have been measured, when the measurement is performed in a random order. However, it may be able to determine how far along the measurement is at any given point, and thus deduce how many blocks have been measured. Given this information, the optimal strategy for the malware is established as follows: malware relocates (copies itself to another block, then resets its former location to a healthy state) at a random location in memory at least once during the measurement of each block. This leads to a probability of escape of $e^{-1} \approx 0.37$ (see [7] for details).

Due to this non-negligible probability, multiple successive and independent measurements are required to achieve reliable detection of malware. With independent measurements, the probability that a piece of malware present on $\mathcal{P}$rv consistently escape detection drops exponentially. For instance, after 13 checks that probability is below 10^{-6}.

SMARM thus results in: (1) increased time to attain a negligible probability of false negatives (due to successive measurements), and (2) additional memory to store the permutation, as compared to techniques using deterministic order with non-interruptibility or non-malleability. However, SMARM can mitigate self-relocating malware for time- and/or safety-critical applications, without resorting to memory locking.

3.3 Self-Measurements

In a typical setting, RA is as an *on-demand* security service: current state of $\mathcal{P}$rv is measured in real-time when a request from $\mathcal{V}$rf is received. Though natural, on-demand RA has two important limitations. First, it is a poor match for unattended devices, since transient malware cannot be detected if it leaves $\mathcal{P}$rv by the

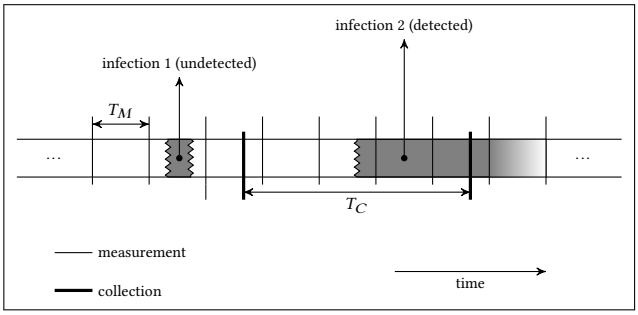

Figure 5: QoA illustration: Infection 1 by transient malware is undetected; Infection 2 is detected. T_M – time between two measurements and T_C – time between two collections.

time attestation is performed. Second, for a device working under real-time constraints (safety-critical operation) on-demand RA is a potentially time-consuming task, which deviates from the device's main function.

To address these limitations, in ERASMUS [6], $\mathcal{P}$rv performs recurrent self-initiated measurements ("self-measurements") that it stores locally. $\mathcal{V}$rf occasionally contacts $\mathcal{P}$rv, collects its stored measurements and verifies them. This highlights two components of the "Quality of Attestation" (QoA) notion: (1) how often $\mathcal{P}$rv's memory is measured, and (2) how often measurements are verified. Figure 5 illustrates these two components, which are conjoined in on-demand RA. However, ERASMUS decouples them and thus offers some advantages. Most importantly, measurements can be performed more often without increased $\mathcal{V}$rf participation. This is an important security consideration, since frequency of (self-)measurements determines the window of opportunity for transient malware. Another advantage is that measurement scheduling can be made context-aware, which is important for safety-critical applications. Although it does not fully resolve the conflict between RA security and critical application needs, ERASMUS offers some possible compromises, e.g., (1) interrupting $\mathcal{M}$P when the application must run; $\mathcal{M}$P can be rescheduled to run thereafter, or (2) adapting $\mathcal{M}$P scheduling such that it does not interfere with application scheduling.

However, ERASMUS does not obviate the need for on-demand attestation. On-demand attestation is still necessary in applications where a quick reaction to infection is crucial; it is the only way to provide the maximum *freshness* for the attestation, given that verification is performed immediately after the measurement. Fortunately, ERASMUS can easily be coupled with on-demand attestation: measurements can be made on $\mathcal{P}$rv based on a schedule *as well as* when receiving a query by $\mathcal{V}$rf.

SeED [14] is an approach similar to ERASMUS. In it, $\mathcal{P}$rv is responsible for initiating the attestation protocol. $\mathcal{V}$rf awaits and verifies attestation responses that it has not initiated. Verifying attestation responses coming from $\mathcal{P}$rv at random times allows $\mathcal{V}$rf to determine the overall integrity of $\mathcal{P}$rv. This approach brings multiple challenges: First, since the response is not bound to a random challenge sent by $\mathcal{V}$rf, it is prone to replay attacks. To overcome this challenge, SeED requires either monotonic counters or synchronized real time clocks. Second, letting the untrusted $\mathcal{P}$rv

determine the attestation time, renders attestation susceptible to transient malware that is capable of disinfecting a malicious $\mathcal{P}$rv right before the initiating of attestation. This challenge can be overcome by keeping attestation time secret from all software running on $\mathcal{P}$rv, i.e., attestation is triggered through a dedicated timeout circuit that has exclusive access to the clock. Third, mitigating a communication adversary that is capable of dropping malicious attestation responses requires $\mathcal{V}$rf know when it expects to receive an attestation response. This can be done by sharing a short random seed between $\mathcal{P}$rv and $\mathcal{V}$rf.

Lack of interaction makes SeED inherently resilient to DoS attacks, which aim at exhausting $\mathcal{P}$rv's resources and prevent it from performing its tasks. This feature is of particular importance in safety-critical applications. Furthermore, SeED improves the efficiency of RA due to its low communication overhead and low network congestion. This improvement leads to a decrease in power consumption of $\mathcal{P}$rv.

In order to provide security against communication and transient malware, SeED requires multiple additional assumptions and hardware components that are not necessary for conventional RA. In particular, synchronized real time clocks might not be available on low-end $\mathcal{P}$rv-s. On the other hand, since the communication in SeED is unidirectional false positives caused by network portioning or communication problems can be problematic as there is no mean of acknowledging the receipt of an attestation response.

4 CONCLUSIONS

In this paper we identified some challenges that stem from the conflict between security requirements of RA and safety-critical application needs. Current techniques require atomic $\mathcal{M}$P execution on $\mathcal{P}$rv. We showed that, since RA can be time-consuming, it can interfere significantly with $\mathcal{P}$rv availability, thus dangerously hampering safety-critical applications. On the other hand, making $\mathcal{M}$P interruptible can lead to failure to detect self-relocating and/or transient malware.

We examined recent techniques that aim to resolve this conflict. They fall into two categories: (i) periodic self-measurements and (ii) interruptible attestation modality that involves shuffled memory traversals and various memory locking mechanisms. Such mitigation techniques offer tradeoffs between malware detection guarantees, temporal consistency guarantees and performance overhead. We believe that these techniques collectively represent the first step towards practical RA in safety-critical embedded devices and a foundation for subsequent research in this area.

SUPPORT: UC Irvine authors' work was supported in part by DHS, under subcontract from HRL Laboratories, and ARO under contract: W911NF-16-1-0536, as well as NSF WiFiUS Program Award #: 1702911. Dr. Sadeghi's work was supported by German Science Foundation CRC 1119 CROSSING S2 project.

REFERENCES

[1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security Symposium*, 2017.

[2] N Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. SEDA: Scalable embedded device attestation. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[3] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: tiny trust anchor for tiny devices. In *ACM/IEEE Design Automation Conference (DAC)*, 2015.

[4] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Lightweigh swarm attestation: a tale of two LISA-s. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[5] Xavier Carpent, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Temporal consistency of integrity-ensuring computations and applications to embedded systems security. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[6] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. ERASMUS: Efficient remote attestation via self-measurement for unattended settings. In *Design, Automation and Test in Europe (DATE)*, 2018.

[7] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. Remote attestation of iot devices via SMARM: Shuffled measurements against roving malware. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2018.

[8] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[9] Hardkernel co. Ltd. ODROID-XU4, 2013.

[10] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Fusing hybrid remote attestation with a formally verified microkernel: Lessons learned. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2017.

[11] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. HYDRA: Hybrid design for remote attestation (using a formally verified microkernel). In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[12] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[13] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. DARPA: Device attestation resilient to physical attacks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2016.

[14] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. SeED: secure non-interactive attestation for embedded devices. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[15] Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher. Standard, ISO.

[16] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[17] Jakob Jonsson, Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. Pkcs# 1: Rsa cryptography specifications version 2.2. 2016.

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *ACM symposium on Operating systems principles*, 2009.

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

[20] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[21] Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, 2010.

[22] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[23] Ahmad-Reza Sadeghi, Matthias Schunter, Ahmad Ibrahim, Mauro Conti, and Gregory Neven. SANA: Secure and scalable aggregate network attestation. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[24] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, 2004.

[25] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Scuba: Secure code update by attestation in sensor networks. In *ACM workshop on Wireless security*, 2006.

[26] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles*, 2005.

[27] Trusted Computing Group. Trusted platform module (tpm).

[28] Jaikumar Vijayan. Stuxnet renews power grid security concerns, june 2010.