Temporal Consistency of Integrity-Ensuring Computations and Applications to Embedded Systems Security

Xavier Carpent
University of California, Irvine
xcarpent@uci.edu

Karim Eldefrawy
SRI International
karim.eldefrawy@sri.com

Norrathep Rattanavipanon
University of California, Irvine
nrattana@uci.edu

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

ABSTRACT

Assuring integrity of information (e.g., data and/or software) is usually accomplished by cryptographic means, such as hash functions or message authentication codes (MACs). Computing such integrity-ensuring functions can be time-consuming if the amount of input data is large and/or the computing platform is weak. At the same time, in real-time or safety-critical settings, it is often impractical or even undesirable to guarantee atomicity of computing a time-consuming integrity-ensuring function. Meanwhile, standard correctness and security definitions of such functions assume that input data (regardless of its size) remains consistent throughout computation. However, temporal consistency may be lost if another process interrupts execution of an integrity-ensuring function and modifies portions of input that either or both: (1) were already processed, or (2) were not processed yet. Lack of temporal consistency might yield an integrity result that is non-sensical or simply incorrect. Such subtleties and discrepancies between (implicit) assumptions in definitions and implementations can be a source of inconsistenceies, which might lead to vulnerabilities.

In this paper, we systematically explore the notion of temporal consistency of cryptographic integrity-ensuring functions. We show that its lack in implementations of such functions can lead to inconsistent results and security violations in protocols and systems using them, e.g., remote attestation, remote updates and secure resets. We consider several mechanisms that guarantee temporal consistency of implementations of integrity-ensuring functions in embedded systems with a focus on remote attestation. We also assess performance of proposed mechanisms on two commodity hardware platforms: I.MX6-SabreLite and ODROID-XU4.

KEYWORDS

embedded system security; remote attestation; temporal consistency

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196526

1 INTRODUCTION

Computation over a large amount of input data is never instantaneous. Even if input size is moderate, computation can take a long time, e.g., if it involves cryptographic primitives, or takes place on a slow (low-end) processor. Assuring atomicity (i.e., uninterruptibility) of computation might be impractical or even unsafe if the underlying system provides critical or real-time service. Meanwhile, if computation is cryptographic in nature and its purpose is to ensure integrity, the result must be *temporally consistent*. In other words, it must, at least[1], reflect the exact state of input data at some point in time. These two requirements are potentially conflicting: if integrity-related computation is interruptible, its input might change, such that the result is inconsistent (i.e., wrong) or non-sensical, i.e., it might correspond to the state of input that did not exist at any one time. This issue has been surprisingly under-appreciated in the security research literature.

More generally, we argue that temporal consistency is important in computing any *integrity-ensuring* function, e.g., checksums for error detection, and not only security-relevant ones such as hash functions, MACs and digital signatures. All these functions are designed to operate on static input data, which is assumed by their standard (security) definitions.

This discrepancy between (implicit) theoretical assumptions and implementations is especially relevant in the context of Remote Attestation (RA). RA is a security service for remotely assessing integrity of software and memory (as well as other types of storage) in embedded devices. RA is typically realized as an interaction between a trusted entity (*verifier*) and an untrusted, potentially malware-infected, remote device (*prover*). Upon a request by verifier, prover computes a measurement of its internal state and returns the result to verifier for validation. The measurement procedure is essentially an integrity-ensuring function with additional security (particulars of which depend on the specific flavor of RA) to prevent malware from falsifying results. Consistency is of paramount concern for RA, since a measurement result must faithfully reflect the state of prover's memory at *some point*. (NOTE: Hereafter, we use *consistency* as a shorthand for *temporal consistency*). Looking at prior RA literature, it is unclear exactly at what time – or time interval – this must hold:

(1) Time when verifier's request is sent to prover?
(2) Time when verifier's request is received by prover?

[1]We say "at least" to mean that the definition of temporal consistency can be expanded to encompass an interval of time, rather than a single point in time.

(3) Time at prover at the very start of its measurement?
(4) Time at prover at the very end of its measurement?
(5) Any time (or interval) between the last two?
(6) The entire period between start and end measurement?

Although this list is not exhaustive, it enumerates the obvious choices.

As an illustrative example, consider a sensor/actuator fire alarm application running on "bare-metal" in a low-end embedded device. This application periodically checks the value of a sensor and triggers an alarm whenever that value exceeds a certain threshold. Given its safety-critical function, software integrity of this device is periodically checked using RA. Upon receipt of a request from the verifier, the measurement process interrupts the application and takes over. The measurement process must run uninterrupted in order to accurately reflect current state of prover's software. One obvious downside of uninterrupted measurement is that the critical application is dormant during this process, even if a real fire occurs.

Whereas, if we favor the critical application and allow the measurement process to be interrupted, another problem arises. Suppose that the device is infected by *migratory malware* – the type of malware that can move itself around – as a whole, or in pieces – in device's memory and other storage, in order to evade detection. Such malware can interrupt the measurement process, e.g., halfway through, and move itself (by copying and erasing) to segments of memory that have been already covered by the measurement process. This way, the final measurement result would reflect a benign (malware-free) state and, upon receiving and checking it, the verifier would not detect any malware presence. For a more detailed discussion of migratory malware, we refer to Appendix D and E.

Although dangers of migratory malware were anticipated in the design of some software-based attestation methods, e.g., Viper and Pioneer [21, 34], tradeoffs between uninterruptibility (and atomicity) and integrity measurement consistency have not been considered in hardware and hybrid attestation designs. Despite their drawbacks, software-based attestation techniques are inherently less vulnerable to migratory malware, since their measurement process involves precise timing which would be noticeably skewed by migratory malware (due to the latter's efforts of copying and erasing). However, as we discuss later, they are also unsuitable for *remote* attestation where fluctuating network delays influence overall timing. Thus, the main goal of this paper is to (1) investigate uninterruptibility/consistency tradeoffs, and (2) design techniques offering a range of concrete consistency guarantees for integrity-ensuring computations, while allowing varying degrees of interruptibility.

Contributions: This paper makes several advances:

(1) First systematic study of temporal consistency in cryptographic integrity-ensuring functions. We show that lack thereof can yield incorrect (including malicious) or nonsensical results.
(2) Design and evaluation of several mechanisms that ensure temporal consistency in the context of embedded systems, with a focus on applicability to secure remote attestation.
(3) As part of this work, we develop a new security game that captures temporal consistency in the context of remote attestation. This security definition may be of independent interest. (See Appendix A).

Outline: Section 2 overviews remote attestation and discusses the importance of temporal consistency. Section 3 introduces our model and notation as well as supporting mechanisms. Section 4 describes several techniques to ensure temporal consistency in remote attestation for embedded and IoT devices. Section 5 describes implementation and performance evaluation of mechanisms proposed in Section 4. Section 6 discusses related work, and Section 7 concludes the paper.

2 TEMPORAL CONSISTENCY

State-of-the-art in stealthy malware has been advancing at an impressive rate. Malware that erases itself after performing an intended task, typically after stealing credential or financial assets, has been discovered in recent years [39]. Malware that utilizes resources (CPU and GPU) on personal computers for computationally heavy (e.g., cryptographic) tasks, mainly to mine cryptocurrencies, has also been reported [25]. Sophistication of malware has increased even more in the realm of Cyber-Physical Systems (CPS), Embedded Systems (ES), and, most recently, Internet-of-Things (IoT). Notable examples include Stuxnet [20, 36] and Duqu [6]. A recent SANS Institute survey [16] about IoT threat vectors and concerns lists malware as the second most highly cited concern (26%), the main justification being fear of IoT devices spreading malware into enterprises. The first concern (31%) was patching and updating software, and the third was denial-of-service (13%).

2.1 Remote Attestation

In recent years, Remote Attestation (RA) emerged as a distinct security service for detecting malware on CPS, ES and IoT devices. RA involves verification of current internal state (i.e., RAM or flash) of an untrusted remote hardware platform (prover or $\mathcal{P}$rv) by a trusted entity (verifier or $\mathcal{V}$rf). RA can help the latter 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 [33] and secure deletion [28]. Many RA techniques with different assumptions, security features and complexities, have been proposed for the single-prover scenario.

Prior RA results can be divided into three approaches: hardware-based, software-based, and hybrid. Hardware-based approaches typically rely on security provided by a Trusted Platform Module (TPM) [15]. Despite resisting all, except physical, attacks, the hardware-based approach is not suitable for low-end and legacy embedded devices due to its added complexity and costs.

Software-based RA techniques offer a very low-cost alternative. Pioneer [34] is a prominent example of this approach. Its main tool is the use of a one-time special checksum function that covers memory (to be attested) in an unpredictable (rather than contiguous) fashion. Any interference with (or emulation of) the computation of this checksum is detectable by extra latency that would be incurred by migratory malware 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., [5]) were demonstrated. Another problem with the software-based approach is its strong

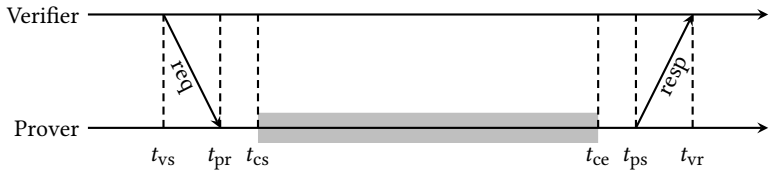


Figure 1: Timeline for a typical remote attestation scheme. Verifier's request is sent at t_{vs} and received at t_{pr}. Computation starts at t_{cs} and ends at t_{ce}. Report is sent at t_{ps} and received at t_{vr}.

assumptions about adversarial capabilities, which are unrealistic in many real networked settings. However, it is the only attestation option for legacy devices.

Hybrid (software-hardware) RA co-designs have been proposed to overcome limitations of purely software-based techniques. SMART [11] is the first hybrid RA architecture with minimal hardware modifications to existing micro-controller units (MCUs). In addition to requiring uninterruptible non-malleable attestation code and attestation keys in read-only memory (ROM), SMART requires hard-wired MCU access control rules to allow access to secret keys only to SMART attestation code. Attestation is performed within $\mathcal{P}$rv's ROM-resident attestation code by computing a cryptographic checksum (e.g., an AES-based CBC-MAC or an SHA2-based HMAC) over a memory region and returning the result to $\mathcal{V}$rf. Notably, SMART requires atomic (uninterruptible) execution of its ROM-resident attestation code. However, this design feature was motivated by the need to mitigate code-reuse attacks (such as ROP [29]) and *not* by consistency of computing the measurement. Follow-on designs, such as TrustLite [19] and TyTAN [2], enhance SMART with secure interrupt handling.

In this paper, we assume that the measuring process ($\mathcal{M}$P) on $\mathcal{P}$rv is realized as a keyed integrity-ensuring function computed over a part (or all) of $\mathcal{P}$rv's memory, in a "protected" execution environment. Exact protection depends on the specific security architecture.

2.2 RA Blueprint

A typical RA scheme operates as follows:

(1) $\mathcal{V}$rf sends a challenge-bearing attestation request to $\mathcal{P}$rv at time t_{vs}
(2) $\mathcal{P}$rv receives it at time t_{pr}
(3) Computation of $\mathcal{M}$P starts at time t_{cs}
(4) Computation of $\mathcal{M}$P ends at time t_{ce}
(5) $\mathcal{P}$rv sends the attestation report to $\mathcal{V}$rf at time t_{ps}
(6) $\mathcal{V}$rf receives it at time t_{vr}

The timeline for this sequence of events is shown in Figure 1. Computation of $\mathcal{M}$P (in gray) may be deferred due to networking delays, $\mathcal{V}$rf's request authentication, or termination of the previously running task. However, typically, $t_{pr} \approx t_{cs}$ and $t_{ce} \approx t_{ps}$. Also, $\mathcal{P}$rv has no control over t_{vs} and t_{vr}. Consequently, hereafter we only consider $t_s := t_{cs}$ (with $t_{cs} = t_{pr}$) and $t_e := t_{ce}$ (with $t_{ps} = t_{ce}$).

As discussed in Section 1, $\mathcal{M}$P may require time-consuming computations. The exact time it takes depends on the size of $\mathcal{P}$rv's memory, its computational capability, and the underlying cryptographic function(s). As a sample hardware platform, we consider $\mathcal{M}$P running an ODROID-XU4 [7] – a single-board computer representative of medium-to-low-end embedded systems. In most cases, (keyed) hashing[2] is the dominant computation, unless memory to be attested is very small, or the signature algorithm is particularly expensive. Figure 2 shows the costs of these operations, for various attested memory sizes and cryptographic algorithms[3]. Above 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 reasonable amount of memory incurs a significant delay. For example, it takes about 0.9s to measure just 100MB on ODROID-XU4. Its entire RAM (2GB) can be measured in about 14s. In a safety-critical setting, this is definitely too long for $\mathcal{M}$P to run uninterrupted.

As mentioned earlier, recent hybrid RA architectures, such as TrustLite [19] and TyTAN [2], 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, 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 $\mathcal{P}$rv is infected with migratory 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 [2], 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. While this protects against a single-process malware from moving in memory, malware that is spread over several colluding processes can defeat this counter-measure. 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 uninterruptibility.

SMART [11] disables interrupts as the first step in $\mathcal{M}$P. This precludes migratory malware. Uninterruptibility is required as a means to protect the attestation key and to ensure $\mathcal{M}$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.

[2]Or encryption for CBC-MAC.
[3]For HMAC, the cost of the second hash is negligible compared to hashing data. Signature time is independent of data sizes, since only the hash of the data is signed.

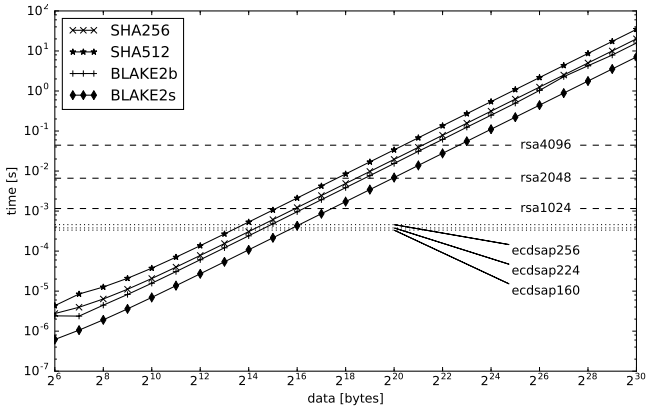


Figure 2: Computational costs of several hash functions and digital signatures on ODROID-XU4.

2.3 A Trivial Approach

One trivial and intuitive way to address the contradicting requirements of temporal consistency and safety-critical operation is to first *copy* memory to be attested over to an area to which $\mathcal{M}$P has exclusive write access. This way, computation can be performed on the copy and $\mathcal{M}$P can be arbitrarily interrupted. This would presumably maximize availability while providing temporal consistency.

Unfortunately, this simple mechanism prompts some concerns. First, it requires sufficient additional memory, which may or may not be available. Second, it requires this additional memory to be locked (either permanently or on demand) to allow $\mathcal{M}$P exclusive write access. Third, copying represents an extra step, which results in longer delays. Finally, it does not fully address the interruptibility/atomicity conflict; it just makes it smaller. Indeed, if copying is uninterruptible, the same time-critical issues can arise, while if interrupts are allowed, migratory malware can, in principle, still evade detection. This is further discussed in Section 4.3.

In the remainder of this paper, we identify and evaluate other mechanisms that reconcile temporal consistency with interruptible execution of $\mathcal{M}$P.

2.4 Attestation Target

The usual target of attestation on $\mathcal{P}$rv is executable code. This code can reside in RAM or in some non-volatile memory. Sometimes, it might also be desirable to attest non-executable regions on $\mathcal{P}$rv (i.e., data) .

Let M, of bitsize L, represent $\mathcal{P}$rv's memory to be attested. If the reference content of M is *a priori* known 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, who can easily validate it. (This is the case if M is supposed to store static application code.) The same applies if M is mutable and its entropy is low: $\mathcal{V}$rf can compute (or pre-compute) all possible valid outputs of $\mathcal{M}$P over M and thus validate $\mathcal{P}$rv's result.

However, if entropy of M is high, enumeration of possible valid states by $\mathcal{V}$rf can quickly become infeasible. This is likely to occur when memory to be attested includes data regions, such as program stack, heap or various registers.

One obvious means of dealing with this problem 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 result of running $\mathcal{M}$P over M, accompanied by a copy of D. Clearly, this only makes sense if D is of modest size, e.g., $|D| << L$.

Furthermore, if D is a highly variable region content of which is either irrelevant or must be empty, $\mathcal{P}$rv can easily zero it out before executing $\mathcal{M}$P. This makes it impossible for malware to hide in such a region and obviates the need for $\mathcal{P}$rv to send $\mathcal{V}$rf an explicit copy of D.

In the remainder of this paper, issues stemming from attestation of static or dynamic memory regions are orthogonal to our work, and thus are not discussed further.

3 MODELING TEMPORAL CONSISTENCY

We now introduce the model and notation for temporal consistency and supporting mechanisms. Although we focus on RA, the model is generic and relevant to other application domains that involve integrity-ensuring functions. In addition to this section, we develop in Appendix A a new definition for a security game that captures temporal consistency in the context of secure remote attestation; we believe that this definition may be of independent interest for future research in remote attestation

We assume that input data is located in $\mathcal{P}$rv's memory M, which consists of n contiguous blocks $[M_1 \ldots M_n]$. Without loss of generality, we assume that block bit-size matches that of the integrity-ensuring function F, e.g., 512 for SHA2-HMAC, or 128 for AES-CBC-MAC. We use M_i to denote content of the i-th block and M_i^t – content of M_i at time t.

We consider computation of $R = F(M)$. For now, we focus on temporal consistency for *sequential* functions, i.e., each M_i is read and processed once during the execution of F and blocks are processed in order: $M_1, M_2, \ldots, M_n$. We model a sequential function F as n independent functions F_i, operating on n blocks sequentially.

Content of memory blocks may change during execution of F, i.e., it might be that $M_i^t \neq M_i^{t'}$ for $t < t'$. However, fetching M_i (to be processed by F_i) is considered to be an atomic operation.

We define temporal consistency for integrity-ensuring functions as follows:

Definition 1. *Output R of an integrity-ensuring function F is* consistent *with input M at time t iff: $R = F(M^t)$.*

We consider F to be *correct and benign*, i.e., it faithfully computes what it is supposed to compute, and its implementation is bug-free. In the context of RA, this holds since $\mathcal{M}$P (containing F) is protected by the underlying security architecture. For example, in hybrid RA architectures, such as TrustLite, TyTAN and SMART, $\mathcal{M}$P is stored in, and executed from, ROM.

We now consider two specific types of malware.

Definition 2. Migratory malware *is present in one or more blocks of M at t_s. It can move (by copying and erasing) itself at any point during computation of F. Its purpose is to remain in M at t_e while remaining undetected.*

Definition 3. Transient malware *is present in one or more blocks of M at time t_s. It can erase itself at any point during computation of F. Its purpose is to escape detection.*

If R is consistent with M at a given time t, and if R corresponds to a benign state, it is guaranteed that no malware was present at time t. This implies that, if $t_s \leq t \leq t_e$, migratory malware cannot escape detection. Furthermore, if $t = t_s$, neither can transient malware.

4 TEMPORAL CONSISTENCY MECHANISMS

We now describe and analyze several mechanisms that offer various tradeoffs between consistency guarantees and real-time requirements. Consistency is achieved through *locking* memory regions, i.e., making them temporarily read-only. Such 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 the formally-verified seL4 [18] microkernel.

Three points in the timeline of computation of an integrity-ensuring function F are particularly relevant to our discussion (see also Figure 3):

(1) t_s, the instance where the computation of F *starts*;
(2) t_e, the instance when the computation *ends*;
(3) Optionally, t_r when $\mathcal{P}$rv is explicitly requested to *release* an existing lock. This release request might come from $\mathcal{P}$rv itself, for instance if R is no longer relevant.

4.1 Simple Approaches

We begin with three obvious options.

4.1.1 `No-Lock`. The simplest mechanism is a strawman that does not lock memory. The result is computed using contents of each memory block M_i at the time when F_i processes it, which means that it provides no consistency guarantees. Consequently, it might not detect migratory or transient malware; see Table 1.

4.1.2 `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. R 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.

4.1.3 `All-Lock-Ext`. An extended variant of `All-Lock` that provides extra consistency keeps all memory locked until t_r. Similar to `All-Lock`, R 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 the verifier 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".

4.2 Sliding Locks

A natural next step for ensuring temporal consistency is to implement "sliding" mechanisms to dynamically lock or unlock blocks of memory during execution of F. Variations of this mechanism are described below and pictured in Figure 4.

4.2.1 Decreasing Lock (`Dec-Lock`*).* This is a less restrictive version of `All-Lock`, which still provides strong consistency guarantees. Entire M is locked at t_s, and each M_i is released as soon as F_i completes processing it. The output R is consistent with all of M at time t_s only. This implies detection of any malware present in M at t_s.

Let t_i be the time that F_i starts/that M_i is loaded. We have the additional guarantee that M_i remains constant between t_s and t_i. It is therefore beneficial to start the computation of F with memory blocks availability of which (to other processes) is important.

4.2.2 Increasing Lock (`Inc-Lock`*).* This variant is the opposite of `Dec-Lock`. The main idea is to lock blocks as they are processed. With entire M unlocked at t_s, it becomes gradually locked as computation of F proceeds, until it is completely locked at t_e, after which it is fully released. Each M_i is locked only when it is time for F_i.

Output R in this case is consistent with M at t_e only. This implies detection of migratory, though not transient, malware. Also, M_i remains constant between t_i and t_e. Unlike `Dec-Lock`, it is beneficial to finish computing F with blocks that require high availability, since they are locked for the shortest time.

As discussed in Section 4.4.1, `Inc-Lock` is better-suited for handling non-sequential functions. On the other hand, locking M can influence the value of the end-result R. In contrast, `Dec-Lock` guarantees consistency at t_s when locking has no impact on R. We consider this to be a subtle yet important distinction between `Dec-Lock` and `Inc-Lock`. Put another way, since `Dec-Lock` does not interfere with any process until t_s, the result R over the snapshot of M at t_s is in no way influenced by the computation of F. However, `Inc-Lock` gradually locks memory and any process that interrupts the execution of F may or may not have write access to parts of memory that it needs: the farther along is the computation of F, the less memory is left unlocked (writable).

4.2.3 Extended Increasing Lock (`Inc-Lock-Ext`*).* 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. R thus remains additionally consistent with M within the interval $[t_e, t_r]$, and M stays constant in $[t_e, t_r]$. This type of extension is not naturally applicable to `Dec-Lock` since memory is not locked at t_e.

Table 1: Malware detection features.

	Migratory Malware	Transient Malware
`No-Lock`	✗	✗
`All-Lock`	✓	✓
`Dec-Lock`	✓	✓
`Inc-Lock`	✓	✗
`Cpy-Lock`	✓	✓

4.3 Mixing Copying with Locking

To minimize the impact on time-critical tasks, M can be first copied to M' and computation of F can be performed with the latter as input. This approach is described below and shown in Figure 5.

4.3.1 Copy Lock (`Cpy-Lock`*).* `Cpy-Lock` reduces the time M is locked by first cloning it and running F over the copy. A lock on M is acquired at t_s and M is copied to another memory segment, M', which is also locked. M' may be a pre-locked portion of memory allocated to F, or a lock on it may be acquired at t_s. Once copying is finished at time t_c, M is entirely free. The second step is to proceed to computing $R = F(M')$.

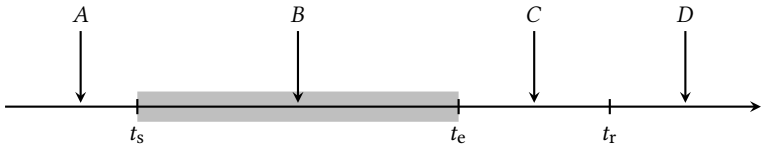


Figure 3: Timeline for computation of $R = F(M)$. Computation starts at t_s and ends at t_e. Consistency of R is considered until t_r. A change to M at time A or D has no effect. Impact of a change at time B or C depends on the consistency mechanism.

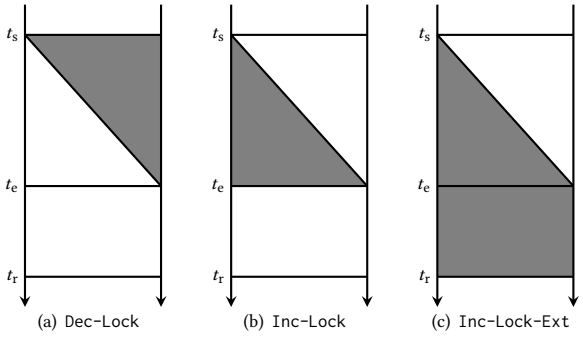


Figure 4: Sliding mechanisms discussed in Section 4.2. M is represented horizontally. Locked portion of M is in gray.

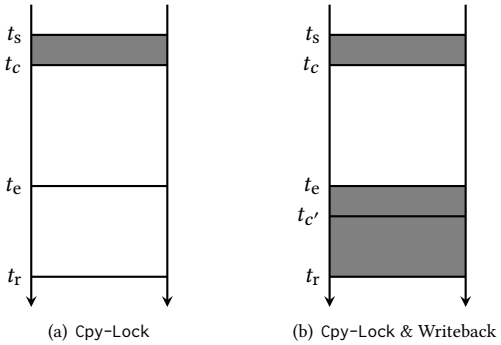


Figure 5: Mechanisms discussed in Section 4.3. M is represented horizontally. Locked portion of M is in gray.

The same guarantees as All-Lock apply here: R is consistent with M in $[t_s, t_c]$.

Cpy-Lock only makes sense if $t_c < t_e$, i.e., if computation of F is more time-consuming than copying M. Depending on how memory locking and unlocking is implemented, it might be better to use Dec-Lock during the copy, instead of All-Lock. Even though the process is less streamlined and possibly less efficient, it may be friendlier towards real-time write requirements on M. Likewise, it is possible to dynamically acquire and release the lock on M' if it is not entirely allocated to F.

4.3.2 Cpy-Lock *& Writeback.* To extend consistency until t_r, one can copy M' back to M once computation of $F(M')$ is finished. M is locked at t_e until t_r. This way, R is consistent with M within the intervals $[t_s, t_c]$ and $[t_e, t_r]$. Consequently, $M^{t_s} = M^{t_e}$, and M remains constant between t_e and t_r. Similar to Cpy-Lock, it might be less constraining to use Dec-Lock during the copy and Inc-Lock during the writeback, instead of All-Lock.

4.4 Variations on the Theme

We outline some extensions to previously discussed mechanisms.

4.4.1 Non-Sequential Functions. Some functions are not sequential, e.g., they might require input blocks to be used concurrently or might reuse blocks in computation. Simple mechanisms (No-Lock or All-Lock) are not affected by this. However, dynamic locking techniques need to be amended.

A lock on M_i needs to be acquired the first time that block is needed by F. Likewise, a lock on M_i can only be released when M_i is no longer required. Consequently, in non-sequential functions, locks may be acquired sooner, or released later, than in sequential

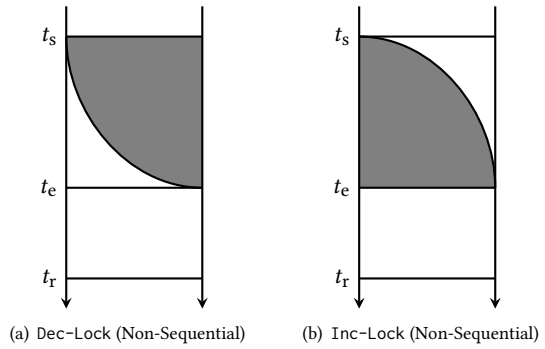


Figure 6: Locked memory in sliding mechanisms of Section 4.2 for non-sequential F. Time goes from top to bottom, and M is represented horizontally. Locked portion of M is in gray.

functions. Figure 6 shows the effect on Dec-Lock and Inc-Lock. A larger gray area indicates more restrictive operation for real-time systems (for the same guarantees of consistency), though still less restrictive than the All-Lock.

Dec-Lock requires the execution environment to be aware of blocks that are no longer needed for the remainder of computation of F. If that information is not available, locks cannot be released until t_e, in which case Dec-Lock degenerates to All-Lock. Inc-Lock does not have this issue (blocks are locked the first time they are needed for F and not freed until t_e).

4.4.2 Adaptive Locking. Multiple mechanisms can be combined in order to achieve alternative timings of consistency in computing

F. For example, to achieve consistency at t_k ($t_s \leq t_k \leq t_e$), we can combine the use of: (1) `Inc-Lock` on $[M_0, \ldots, M_k]$, and (2) `Dec-Lock` on $[M_k, \ldots, M_n]$. Nevertheless, it is somewhat unclear if and when such hybrids may be useful in practice. One potentially relevant application is *adaptive locking* that aims to minimize impact on other processes, especially, if the execution environment is aware of other processes' interrupt schedules.

4.4.3 Lazy Copy (`Cpy-Lazy`*).* Another variation of copy-based mechanisms in Section 4.3 is `Cpy-Lazy`. It involves using `All-Lock`[4] on M with a lazy (or reactive) copy mechanism. When another process interrupts F and, during its execution, wishes to write M_i, this block is first copied to M'_i. The lock on M_i is then released so the process can write to it. The rationale for `Cpy-Lazy` is that copying only what is, and when, necessary reduces overhead. This is particularly relevant when few blocks are likely to be modified during computation of F. However, if many blocks are to be modified and copied, cumulative overhead might exceed that of a single bulk copy. Another consideration is whether there is OS or hardware (e.g., MPU) support for the "interrupt-on-write" primitive required to implement `Cpy-Lazy`.

4.5 Uninterruptibility *vs.* Locking

All mechanisms described above achieve consistency by temporarily locking (parts of) memory. As mentioned earlier, uninterruptibility of computation of F (e.g., as in SMART [11]) also provides consistency, though rigidly, i.e., for the interval $[t_s, t_e]$. There are other differences:

- Even when M is locked entirely or partially, other processes can interrupt execution of F and modify memory outside of M, as well as read all memory, including M. This does not violate consistency of F's result R.
- Whereas, if F is uninterruptible and the underlying hardware platform is a single-CPU device, other processes are completely blocked, regardless of whether M is locked.
- If multiple CPUs have shared memory access, uninterruptibility **does not** guarantee consistency, since a process running on a CPU different from the one running F can modify M concurrently.
- Locking is more flexible than uninterruptibility: while locking and unlocking of M can be dynamic and gradual (i.e., block-wise), execution of F is rigid: either it is interruptible or not. For example, SMART provides consistency only because, in a single-CPU device, uninterruptibility is equivalent to `All-Lock`.

4.6 Memory Access Violations

If some process P' tries to write to M_i which is currently locked by process P running F, a memory access violation occurs (recall that read access to M requires no extra handling). P and P' might be running concurrently, on different CPUs, or P' might have interrupted P. There are several alternatives:

If P handles the situation, one possibility is to abort F and terminate P. This approach is the most friendly with respect to P' and other processes. However, it makes it easy for a malicious process to starve P, i.e., prevent F from ever completing. Otherwise, we can adopt the reactive `Cpy-Lazy` approach discussed in Section 4.4.3. Alternatively, P' can be aborted. Though this would allow P (and thus F) to complete uninterrupted, it might be impractical in safety-critical scenarios. Another possibility is to stall P' until M_i is unlocked. This approach is gentler, although it might still be problematic, depending on how long P' has to wait.

4.7 Inconsistency Detection

Another approach to *enforce consistency* is to *detect inconsistency*. The memory M is not locked but instead a trigger is setup such that the integrity measuring (e.g., attestation) process is alerted if any changes occur to M during the computation of F. If any such changes occur, the result produced is thus no longer consistent throughout the computation. Depending on the strategy for dealing with inconsistency, the computation of F can be stopped, continued, or restarted. An implementation of this is presented and discussed in Section 5.5.

The clear benefit of inconsistency detection over consistency enforcement is that it does not interfere with the execution of other processes. This is particularly relevant in time-critical applications when availability must be maintained at all times. The drawback is that consistency might not be guaranteed, depending on the strategy used whenever an inconsistency is detected. This may lead to attestation never terminating if inconsistencies are constantly created, even by benign software.

5 IMPLEMENTATION & EVALUATION

Our prototype of temporal consistency mechanisms is realized in the context of HYDRA hybrid RA architecture [10]. Below, we overview HYDRA, discuss implementation details of each mechanism and assess their performance on two popular low- to medium-end development boards: I.MX6-SabreLite [9] and ODROID-XU4 [7]. Security considerations for our implemented mechanisms are discussed in Appendix B.

5.1 HYDRA

HYDRA implements a hybrid RA design 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. Using the (mathematically) proven isolation features of seL4, access control rules can be implemented in software and enforced by the microkernel. Note that, in addition to the design of seL4 being formally verified and ensured to guarantee isolation, seL4 software implementation is also formally verified for conformance to the design.

HYDRA stores an attestation key ($\mathcal{K}$) and attestation code (that computes a MAC using $\mathcal{K}$) in a writable memory region (e.g., flash or RAM) and configures the system such that no other process, besides the attestation process (P_{Att}), can access this memory region. Access control configuration in HYDRA also involves P_{Att} having exclusive access to its thread control block as well as to memory regions used for $\mathcal{K}$-related computations. The latter ensures that $\mathcal{K}$ is properly protected. To ensure uninterruptibility, HYDRA runs the attestation process as the so-called *initial user-space process* with the highest scheduling priority. As the initial user-space process in

[4] It can also be easily adapted to `Inc-Lock` and `Dec-Lock`.

seL4, P_{Att} is also initialized with capabilities to all memory pages. Meanwhile, the rest of user-space processes are assigned lower priorities and spawned by P_{Att}. Finally, hardware-enforced secure boot feature is used to ensure integrity of seL4 itself and of P_{Att} when the system is initialized.

5.2 Experimental Setup

Our implementation ensures temporal consistency by locking memory regions. It thus does not require the execution of P_{Att} to be uninterruptible, unlike the original HYDRA implementation [10]. As a result, all user-space processes, including P_{Att}, have the same priority in our implementation.

The microkernel executable is compiled from the unmodified seL4 source code v4.0.0 [27]. Our user-space code is based on open-source seL4 libraries [26], mostly for providing abstractions for processes, memory management and virtual address space.

5.3 Experimental Results: Primitives

Our implementation of mechanisms discussed in Section 4 consists of four primitives: *LockPage*, *UnlockPage*, *CopyMem* and *MacMem*. In HYDRA (and in seL4, in general), locking and unlocking a memory page can be invoked from user-space (by authorized processes) and handled inside the kernel.

To lock a specific page, P_{Att} needs to perform three steps: (1) revoke all capabilities associated with the page [5], (2) create a read-only capability to the page, (3) assign the new capability to a targeted process and map the page into the process' virtual address space. Unlocking can be done similarly by using a read-and-write capability, instead of a read-only capability. In terms of seL4 implementation, each of these primitives translates into three function calls: *seL4_CNode_Revoke*(), *seL4_CNode_Copy*() and *seL4_ARCH_Page_Map*().

Another parameter related to *LockPage* and *UnlockPage* is memory page size, which can differ depending on the underlying instruction-set architecture. For instance, I.MX6-SabreLite, which is based on the ARMv7-A architecture, only supports the following page sizes: 4KB, 64KB, 1MB and 16MB. *CopyMem* performs a memory copy between source and destination RAM locations. We note that only `Cpy-Lock` requires this primitive. Finally, *MacMem* performs a MAC computation over a memory range. *MacMem* is implemented as a keyed hash using: BLAKE2S [31], AES256-CBC based MAC [17] and HMAC-SHA256 [37] algorithms.

Figure 7 illustrates run-time of primitive operations on 16MB of memory. Results show that page size heavily influences performance of *LockPage* and *UnlockPage*: the larger the page size, the faster it is to lock or unlock memory of the same size. This is expected, because larger pages result in fewer entries that need to be modified in a page table. Run-time performance of *CopyMem* and *MacMem*, however, remains almost unchanged, regardless of page size. In addition, the same figure suggests that run-times of *CopyMem*, *LockPage* and *UnlockPage* are relatively fast, compared to that of *MacMem*. The first three primitives take at most 9% of *MacMem*'s run-time.

[5] This step by default includes modifying the corresponding page table entry, clearing a cache line and invalidating a TLB entry.

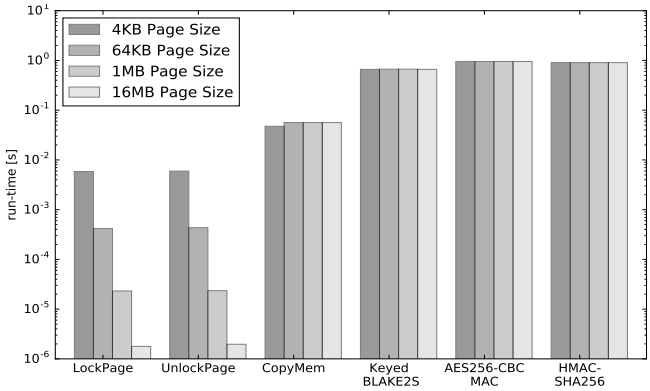


Figure 7: Performance of primitives with 16MB of memory on I.MX6-SabreLite.

Finally, we evaluate and compare performance of the various primitives on I.MX6-SabreLite running at 1.0GHz, and ODROID-XU4 running at 2.1GHz. Figure 8 shows the results of this comparison. It shows that: (1) run-times of *LockPage* and *UnlockPage* primitives are still roughly the same on both hardware platforms, and (2) *MacMem* remains, by far, the most time-consuming primitive.

5.4 Experimental Results: Mechanisms

We assess performance of five temporal consistency mechanisms – `No-Lock`, `All-Lock`, `Dec-Lock`, `Inc-Lock` and `Cpy-Lock` – on the SabreLite board. `No-Lock` is the baseline and it directly translates into the *MacMem* primitive. `All-Lock`, `Dec-Lock` and `Inc-Lock` all require additional steps of sequentially locking and unlocking memory blocks. For its part, `Cpy-Lock` involves all four primitives.

Figure 9 demonstrates run-time performance of aforementioned mechanisms (using BLAKE2S as the underlying function) with various memory sizes: 16MB to 96MB, and page sizes 4KB and 64KB. Results can be summarized as follows:

- Run-time of all mechanisms is linear in terms of memory size. This is expected since they are built upon a sequential function, i.e., a MAC.
- Run-time of MAC computation on large memory sizes is indeed non-negligible, e.g., it takes around 4 seconds for keyed BLAKE2S over 96MB of memory. This clearly demonstrates the need for ensuring temporal consistency, especially, in settings where P_{Att} needs to be interruptible.
- Run-times of `All-Lock`, `Dec-Lock` and `Inc-Lock` are all roughly equal, in all cases. This is also expected, since each of these three mechanisms involves a similar number of invocations of primitives.
- The difference in run-time between baseline and `All-Lock`, `Dec-Lock` and `Inc-Lock` decreases as page size grows. This difference then becomes negligible ($< 0.1\%$) when page size reaches 1MB. Thus, it is beneficial to use these mechanisms with reasonably large page sizes. One disadvantage of larger page sizes is that memory pages, on average, will be locked for longer periods.

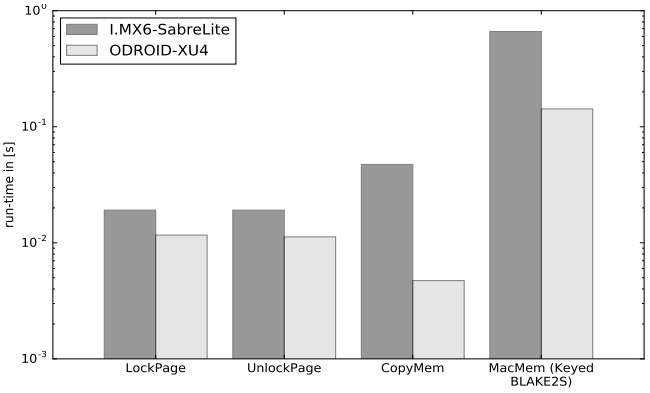


Figure 8: Performance of primitives with 16MB memory on I.MX6-SabreLite and ODROID-XU4.

- `Cpy-Lock` comes out as the preferred mechanism. It incurs small (~ 8%) run-time overhead; however, this mechanism provides much better availability as memory is locked for a very short amount of time (only during the copying process). However, recall that an obvious disadvantage is that it requires additional memory of size M'.

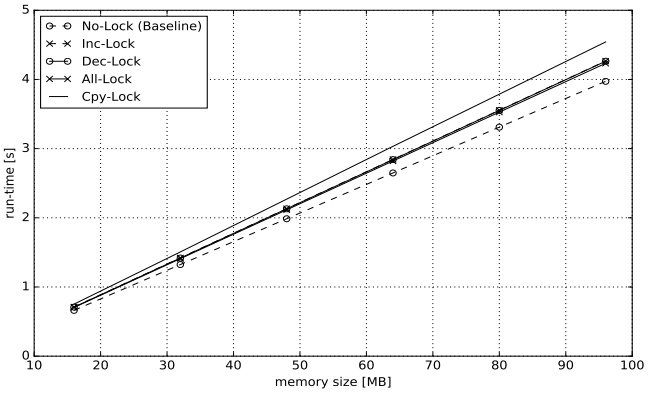
5.5 Implementation of Inconsistency Detection

We could implement the inconsistency detection mechanism by having P_{Att} detect whether any dirty/accessed bits are set after each measurement is completed. However, this obvious approach falls short in the context of HYDRA. Doing so would imply some modifications to the existing kernel, which may consequently break formally verified properties of seL4.

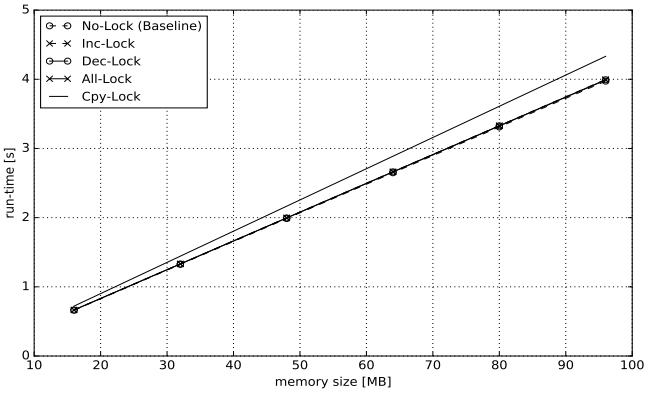
Instead, we base our implementation of inconsistency detection on the `All-Lock` implementation. The idea is to have P_{Att} first lock memory to be attested before starting to compute the integrity-ensuring function, e.g., the MAC. If the computation completes without interruptions or detecting any inconsistency, P_{Att} then unlocks the memory; this scenario resembles typical `All-Lock` execution. However, if another process (denoted by P') attempts to modify any part of the locked memory, the kernel will suspend execution of P' and P_{Att} will be made aware of such inconsistency; P_{Att} then resolves the inconsistency by unlocking the memory and resuming execution of P'. Note that this implementation still requires some interference with other processes as P' is suspended when inconsistency occurs. However, we show later in Section 5.6 that the overhead from this interference is very small compared to the actual measurement runtime.

To implement this mechanism in HYDRA, we decompose P_{Att} into the following three threads:

- $\mathsf{Th}_{\mathsf{checksum}}$: computing the integrity-ensuring function and returning an attestation result to $\mathsf{Th}_{\mathsf{main}}$ on success.
- $\mathsf{Th}_{\mathsf{fault}}$: listening for any memory write fault and notifying $\mathsf{Th}_{\mathsf{main}}$ when there is an attempt to modify memory being attested.
- $\mathsf{Th}_{\mathsf{main}}$: managing the other two threads, locking and unlocking memory and reporting to $\mathcal{V}$rf when an inconsistency occurs.



(a) 4KB Page Size



(b) 64KB Page Size

Figure 9: Run-time of various temporal consistency ensuring mechanisms in I.MX6-SabreLite.

Unlike $\mathsf{Th}_{\mathsf{checksum}}$ and $\mathsf{Th}_{\mathsf{main}}$, implementing $\mathsf{Th}_{\mathsf{fault}}$ is not trivial; it requires support from the underlying hardware and/or kernel in order to: (1) detect whenever a process causes a fault and (2) examine whether the fault is caused by an invalid write access and whether it happens within a specific memory range. Fortunately, these operations are already available in seL4 without requiring modifications to the kernel.

We implement $\mathsf{Th}_{\mathsf{fault}}$ by leveraging how *a fault endpoint* works in seL4. An endpoint is an seL4 object that allows a small amount of data to be transferred between two threads. When a process or a thread faults, the seL4 kernel automatically sends a fault IPC message to its registered fault endpoint. This fault IPC message provides useful information that helps $\mathsf{Th}_{\mathsf{fault}}$ decide whether the fault will result in memory inconsistency. For instance, the message includes a type of fault (e.g. page fault, capability fault, or unknown syscall), address that causes the fault and whether a read or write access causes the fault[6]. In our implementation, $\mathsf{Th}_{\mathsf{main}}$ shares a single fault endpoint among all user-space processes, allowing a fault caused by any process to be transmitted to this fault endpoint. The last step of the implementation is to have $\mathsf{Th}_{\mathsf{fault}}$ wait for an incoming message from the fault endpoint and notify $\mathsf{Th}_{\mathsf{main}}$ if the

[6] See http://sel4.systems/Info/Docs/seL4-manual-latest.pdf for full details.

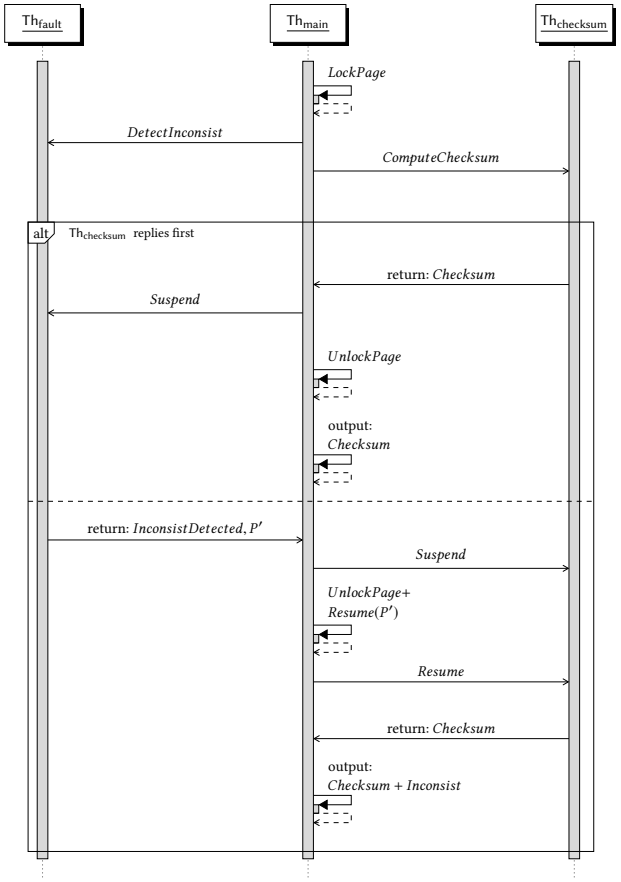


Figure 10: Sequence diagram of P_{Att} with memory inconsistency detection during single attestation instance. P_{Att} chooses to resume execution of $\mathsf{Th}_{\mathsf{checksum}}$ after P' causes memory inconsistency.

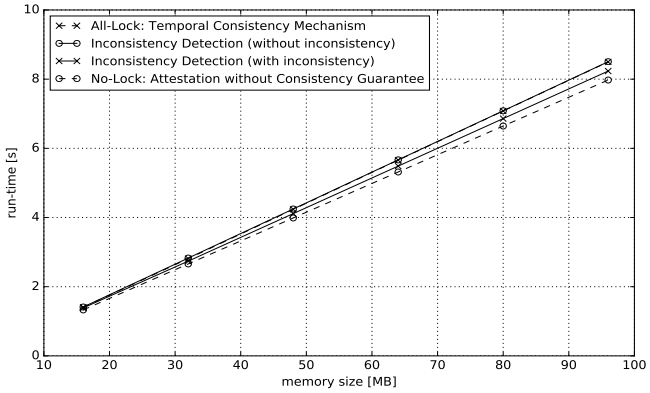


Figure 11: Run-time of inconsistency detection with 4KB page size on I.MX6-SabreLite.

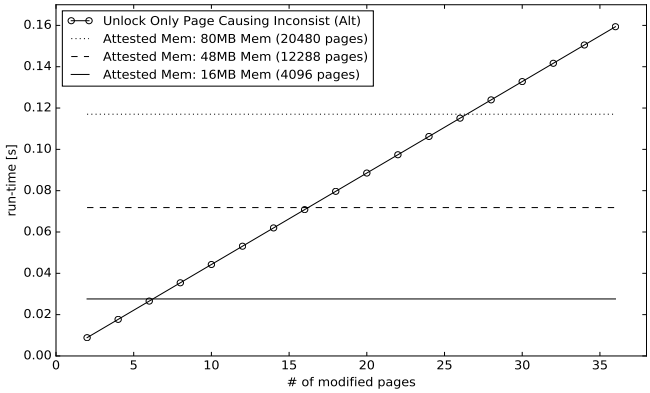


Figure 12: Downtime of the faulting process P' when its actions result in an inconsistency with 4KB page size on I.MX6-SabreLite. Horizontal lines represent downtime from the approach where P_{Att} resolves inconsistency by unlocking entire memory of P'.

message indicates the attempted write access on memory being attested. A sample code for $\mathsf{Th}_{\mathsf{fault}}$ is provided in Appendix C.

A diagram in Figure 10 summarizes the sequence of operation of our modified P_{Att} during a single attestation instance. First, $\mathsf{Th}_{\mathsf{main}}$ locks entire memory to be attested, then calls $\mathsf{Th}_{\mathsf{checksum}}$ and $\mathsf{Th}_{\mathsf{fault}}$ via a shared endpoint and waits for their replies. There are two possible scenarios:

(1) If no process attempts to write into attested memory during attestation, $\mathsf{Th}_{\mathsf{checksum}}$ successfully completes and returns to $\mathsf{Th}_{\mathsf{main}}$ with an attestation token. $\mathsf{Th}_{\mathsf{main}}$ then promptly unlocks attested memory.
(2) Otherwise, the kernel suspends P' and transmits a fault IPC message to $\mathsf{Th}_{\mathsf{fault}}$. Once receiving it, $\mathsf{Th}_{\mathsf{fault}}$ replies back to $\mathsf{Th}_{\mathsf{main}}$, which suspends $\mathsf{Th}_{\mathsf{checksum}}$, unlocks memory, and resumes execution of P'. $\mathsf{Th}_{\mathsf{main}}$ can also choose to abort, continue or restart execution of $\mathsf{Th}_{\mathsf{checksum}}$.

Finally, $\mathsf{Th}_{\mathsf{main}}$ outputs the result (an attestation token and/or whether any inconsistency occurs or not) back to $\mathcal{V}$rf.

5.6 Experimental Results: Inconsistency Detection

To evaluate performance of the inconsistency detection mechanism, we experimented by running two processes – modified P_{Att} and P' – with the same execution priority on I.MX6-SabreLite. (Multiple same-priority processes are scheduled in a round-robin fashion.) Thus, timing results for this experiment differ from others that consider only P_{Att} running at any given time.

Results in Figure 11 show the performance comparison of: (1) the inconsistency detection mechanism (with and without inconsistency occurring), (2) `All-Lock`, and (3) attestation without consistency guarantee or `No-Lock` on 16MB to 96MB memory. In this experiment, we assume that P_{Att} chooses to resolve inconsistency by unlocking the entire memory of P'. In case of no inconsistency, our mechanism (as expected) performs as well as `All-Lock` and roughly 6% slower than `No-Lock`. On the other hand, when an inconsistency occurs, the mechanism (surprisingly) runs 3% faster. While this may seem counter-intuitive, we found that improved performance is caused by $\mathsf{Th}_{\mathsf{main}}$ performing memory unlocking while P' is being suspended. This results in run-time of the unlock

operation being ~2x faster than that of the same operation in its counterpart, where memory unlocking is performed concurrently with P'.

We now consider the alternative, whereby P_{Att} resolves the inconsistency by unlocking only the page that causes it, instead of unlocking entire memory. Clearly, runtime overhead of this approach depends on the number of times inconsistency is triggered [7] during attestation. In this experiment, we measure overhead through *downtime* of P', i.e., total elapsed time for P' to complete writing into locked pages. Figure 12 illustrates that overhead, as expected, is linear in terms of a number of modified pages. It also shows that it is more beneficial to use the alternative approach where P' is expected to perform only a few memory writes. In our experimental setting, this threshold is around 0.12% of P' memory pages.

6 RELATED WORK

To the best of our knowledge, there has been no prior work on temporal consistency of integrity-ensuring functions, though it is possible that this concept has been considered under a different guise in the cryptographic literature. Extended versions `Inc-Lock-Ext` and `Cpy-Lock` & Writeback can be viewed as a form of protection against "Time of Check Time of Use" (TOCTOU) attacks in certain applications.

The "Provable Virus Detection" method discussed in [22] is a very relevant result. In it, a secret is embedded within software running on a device is periodically checked by a trusted verifier. The argument is that injected malware consistently destroys the secret, and its presence is therefore detectable. While promising, [22] only deals with malware directly inserted into a system (e.g., via DMA) and requires substantial modifications to the CPU.

One alternative way to detect malware without locking memory (however, without guaranteed consistency) is explored in [4]. Memory is measured in a random order, which cannot be learned or anticipated by malware. Since memory is never locked, this is an advantage for time-sensitive applications. The main drawback of [4] is its probabilistic nature, which can lead to a significantly increased time to perform attestation.

The rest of this section focuses on related work in RA.

RA aims to detect malware presence by *verifying* integrity of a remote and untrusted embedded (or IoT) device. It is typically realized as a protocol, whereby a trusted verifier interacts with a remote prover to obtain a challenge-based integrity measurement of the latter's memory state. RA techniques fall into three main categories: hardware-based, software-based, and hybrid.

Hardware-based attestation [32, 38] uses dedicated hardware components, such as a Trusted Platform Module (TPM) [15], ARM TrustZone [23] or Intel SGX [8] to execute attestation code in a trusted execution environment. Even though such features are currently available in personal computers, laptops and smartphones, they are still considered a "luxury" for low-end embedded devices.

Software-based attestation [34, 35] requires no hardware support and performs attestation solely based on software and precise timing measurements. When deployed on a single-processor system, this approach can ensure temporal consistency; malware could try to interrupt the measurement process and cause temporal inconsistency (e.g. by moving itself around) during attestation. However, this action will result in additional delay, which is then detectable by $\mathcal{V}$rf. Software-based approaches limit the prover to being one-hop away from the verifier, in order to ensure that the round-trip time is either negligible or fixed. Such approaches also rely on strong assumptions about attackers' behavior [1] and are typically used only for legacy devices, where no other RA techniques are viable.

Finally, *hybrid attestation* [2, 11, 19], based on software/hardware co-design, realizes RA while attempting to minimize required hardware features and the software footprint. SMART [11] is the first hybrid RA design with minimal hardware modifications to existing microcontroller units (MCUs). It has the following key features:

- Attestation code is immutable: it is located in, and executed from, ROM.
- Attestation code is safe: its execution always terminates and leaks no information other than the attestation result.
- Attestation code is executed atomically: (1) it is uninterruptible, and (2) it starts from the first instruction and exits at the last instruction. (This is enforced by hard-wired MCU access controls and disabling interrupts upon entry of attestation code.)
- A secret attestation key is stored in an isolated memory location that can be accessed (based on hard-wired MCU rules) only from within attestation code.

Subsequently, [3] extended SMART to defend against verifier impersonation and denial-of-service (DoS) attacks. The resultant design (SMART+) additionally requires prover to have a Reliable Read-Only Clock (RROC), which is needed to perform verifier authentication and prevent replay, reorder and delay attacks. To ensure reliability, RROC cannot be modified by non-physical (software) means. Upon receiving a verifier request, ROM-resident attestation code checks the request's freshness using RROC, authenticates it, and only then proceeds to perform attestation.

TrustLite [19] security architecture also supports RA for low-end devices. It differs from SMART in two ways: First, interrupts are allowed and are 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 [2], adopts a similar approach while providing additional real-time guarantees and dynamic configuration for safety- and security-critical applications. As mentioned earlier, both TrustLite [19] and TyTAN [2] support interrupts. While this allows for time-critical processes to take priority over others and to preserve $\mathcal{P}$rv's functionality, attestation results may not be consistent. Memory can change once attestation is interrupted and the final attestation result might correspond to a state of $\mathcal{P}$rv's memory that never existed.

In summary, RA architectures that disable interrupts, or ensure atomic execution through other means, automatically (though only coincidentally) ensure temporal consistency on single-processor devices. In multi-processor settings, atomic execution is insufficient. Whereas, RA architectures that allow interrupts must ensure temporal consistency (e.g., via mechanisms described in this paper); otherwise nonsensical or incorrect results might be produced.

[7] This is equivalent to the number of memory pages of P' modified during attestation.

7 CONCLUSIONS

In this paper we explore the discrepancy between (implicit) theoretical assumptions and implementations of cryptographic integrity-ensuring functions, focusing on the context of Remote Attestation (RA). We show that, in practice, inputs to such functions can change during computation, and that the vulnerability window can be large, since cryptographic computations can be time-consuming. We propose multiple practical mechanisms to ensure consistency of integrity-ensuring functions. They offer tradeoffs between consistency guarantees, performance overhead, and impact on memory availability. We implement proposed mechanisms on two commodity platforms in the context of a hybrid RA architecture for embedded systems. Results show that locking/unlocking of memory incurs negligible overhead over computing cryptographic integrity-ensuring functions, e.g., MACs. We demonstrate that ensuring temporal consistency can be achieved with less than 10% overhead on both platforms, while providing much better availability for time-critical applications. We believe that this paper highlights important issues that have been surprisingly under-appreciated in the security research literature, yet are crucial for correct and secure operations in RA and other security services building upon it.

SUPPORT: This work was supported in part by (1) DHS, under subcontract from HRL Laboratories, (2) ARO under contract: W911NF-16-1-0536, and (3) NSF WiFiUS Program Award #: 1702911.

REFERENCES

[1] Tigist Abera, N Asokan, Lucas Davi, Farinaz Koushanfar, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. Invited: Things, trouble, trust: on building trust in IoT systems. In *ACM/IEEE Design Automation Conference (DAC)*.

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

[3] Ferdinand Brasser, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. Remote Attestation for Low-End Embedded Devices: the Prover's Perspective. In *ACM/IEEE Design Automation Conference (DAC)*.

[4] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. 2018. Remote Attestation of IoT Devices via SMARM: Shuffled Measurements Against Roving Malware. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2018*.

[5] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the Difficulty of Software-based Attestation of Embedded Devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*.

[6] Eric Chien, Liam OMurchu, and Nicolas Falliere. 2012. W32.Duqu: The Precursor to the Next Stuxnet. In *Proceedings of the 5th USENIX Conference on Large-Scale Exploits and Emergent Threats*.

[7] Hardkernel co. Ltd. 2013. ODROID-XU4. (2013). http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* (2016).

[9] Boundary Devices. 2017. BD-SL-I.MX6. (2017). https://boundarydevices.com/product/sabre-lite-imx6-sbc/

[10] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2017. HYDRA: Hybrid Design for Remote Attestation (Using a Formally Verified Microkernel). In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.

[11] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *Network and Distributed System Security Symposium (NDSS)*.

[12] F-Secure. 2018. Brain Description. (2018). https://www.f-secure.com/v-descs/brain.shtml

[13] F-Secure. 2018. Cabanas Description. (2018). https://www.f-secure.com/v-descs/cabanas.shtml

[14] F-Secure. 2018. Frodo Description. (2018). https://www.f-secure.com/v-descs/frodo.shtml

[15] Trusted Computing Group. 2017. Trusted Platform Module (TPM). (2017). http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/

[16] SANS Institute. 2014. Securing the Internet of Things Survey. (2014). https://www.sans.org/reading-room/whitepapers/analyst/securing-internet-things-survey-34785

[17] ISO/IEC. 2011. *Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher*. Standard. ISO.

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*.

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

[20] Ralph Langner. 2013. To Kill a Centrifuge a Technical Analysis of What Stuxnet's Creators Tried to Achieve. (2013).

[21] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. 2011. VIPER: Verifying the Integrity of PERipherals' Firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*.

[22] Richard J. Lipton, Rafail Ostrovsky, and Vassilis Zikas. 2016. Provably Secure Virus Detection: Using The Observer Effect Against Malware. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP*.

[23] ARM Ltd. 2017. ARM TrustZone. (2017). https://www.arm.com/products/security-on-arm/trustzone

[24] LWN.net. 2018. DR rootkit released under the GPL. (2018). https://lwn.net/Articles/297775/

[25] Wired Magazine. 2013. Trojan Turns Your PC Into Bitcoin Mining Slave. (2013). https://www.wired.com/2013/04/bitcoin-trojan

[26] National ICT Australia and other contributors. 2014. seL4 Libraries. (2014). https://github.com/seL4/seL4_libs

[27] National ICT Australia and other contributors. 2014. The seL4 Repository. (2014). https://github.com/seL4/seL4

[28] Daniele Perito and Gene Tsudik. 2010. Secure Code Update for Embedded Devices via Proofs of Secure Erasure.. In *ESORICS*.

[29] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* (2012).

[30] Ethan M Rudd, Andras Rozsa, Manuel Günther, and Terrance E Boult. 2017. A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions. *IEEE Communications Surveys & Tutorials* (2017).

[31] MJ Saarinen and JP Aumasson. 2015. *The BLAKE2 cryptographic hash and message authentication code (MAC), RFC 7693*. Technical Report. IETF.

[32] Dries Schellekens, Brecht Wyseur, and Bart Preneel. 2008. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming* (2008).

[33] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2006. SCUBA: Secure Code Update By Attestation in Sensor Networks. In *ACM Workshop on Wireless Security (WiSe)*.

[34] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2005. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*.

[35] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. 2004. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Research in Security and Privacy (S&P)*.

[36] IEEE Spectrum. 2013. The Real Story of Stuxnet. (2013). http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet

[37] Secure Hash Standard. 2002. FIPS PUB 180-2. (2002).

[38] Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. 2006. A Robust Integrity Reporting Protocol for Remote Attestation. In *Workshop on Advances in Trusted Computing (WATC)*.

[39] Symantec. 2015. GreenDispencer: Self-deleting Malware. (2015). https://www.symantec.com/security_response/writeup.jsp?docid=2015-092513-0300-99

[40] Wei Yan, Zheng Zhang, and Nirwan Ansari. 2008. Revealing packed malware. *seCurity & PrivaCy* (2008).

APPENDIX

A TEMPORAL CONSISTENCY SECURITY GAME

We build upon the theoretical model of a processor architecture and syntax from [22]. The work in [22] focuses on virus detection by constructing a scheme that interleaves secret shares of cryptographic keys with the actual memory. This scheme requires modifications to the instructions of the processor, in order to reconstruct such keys and use them to ensure integrity (and thus detect unauthorized modifications by malware) of memory content with every read and write. Our work differs from [22], since we do not require any modification to the underlying processor architecture, as evident in our implementation.

A.1 System (Memory and CPU) Model

We model the prover as a random access machine RAM made up of two components: a random access memory M, and a central processing unit CPU. M consists of three sections:

(1) MEM– standard random access memory.
(2) ROM– read-only memory. This section of memory will store the code for a measuring process $\mathcal{M}$P.
(3) ProMEM– protected memory, that can only be written to from instructions in ROM. This section of memory will store data to be used by the $\mathcal{M}$P in ROM.

CPU consists of registers (including input and output register) and an instruction set. Communication between M and CPU occurs in fetch-execute cycles, which are referred to as *rounds* below.

A.2 Syntax of a Consistent Integrity-Ensuring Measurement Scheme

A consistent integrity-ensuring measuring scheme ($\mathcal{CMP}$) is a tuple of algorithms (Gen, Challenge, Respond, Verify) defined as:

- Gen(λ): Generates a secret key $\mathcal{K}$ on input of a security parameter λ.
- Challenge(s): Generates a random challenge c on input of a seed s.
- Respond(M, c, $\mathcal{K}$): Generates a response r to a given challenge c (based on content of memory M).
- Verify($c, r, \mathcal{K}$): Outputs a bit b indicating whether r is a valid response to the challenge c.

A.3 Consistent Integrity Ensuring Measurement Attack Game

In the following game, $\mathcal{A}$ is allowed to choose a piece of code (or data) to inject into memory at any point in time. At some point in time chosen by $\mathcal{A}$, a challenge is issued. $\mathcal{A}$ wins if its code (or data) is injected before the game ends, but the response to the challenge is correct.

Recall that, in Section 2, we described a typical RA scheme as follows:

(1) $\mathcal{V}$rf sends a challenge-bearing attestation request to $\mathcal{P}$rv at time t_{vs}
(2) $\mathcal{P}$rv receives it at time t_{pr}
(3) Computation of $\mathcal{M}$P starts at time t_{cs}
(4) Computation of $\mathcal{M}$P ends at time t_{ce}
(5) $\mathcal{P}$rv sends the attestation report to $\mathcal{V}$rf at time t_{ps}
(6) $\mathcal{V}$rf receives it at time t_{vr}

The formal security game of $\mathcal{CMP}$ is defined in terms of rounds, where if $t_{\text{vs}} = t_{\text{pr}} = t_{\text{cs}}$, they would all correspond to the instant at the end of the rounds ρ_{init} when the challenge is issued. The end of ρ_{attest} corresponds to time when computation of the integrity ensuring function ends at: $t_{\text{ce}} = t_{\text{ps}} = t_{\text{vr}}$.

Table 2: Notation

$\mathcal{A}$	The adversary
C	The challenger
ρ_{init}	# rounds at beginning of security game (before issuing challenge)
ρ_{insert}	# rounds before $\mathcal{A}$'s code is injected
ρ_{attest}	# rounds after issuing the challenge
v	Code that $\mathcal{A}$ injects into MEM
$\mathcal{M}$P	Integrity-ensuring measurement function that runs Respond algorithm.

Definition 4. *We say that a consistent integrity-ensuring measuring scheme ($\mathcal{CMP}$) is* ***secure*** *if a non-empty piece of code is inserted before the attack game terminates, and:*

$$Pr(b = 1) \leq \mu(\lambda)$$

where $\mu(\lambda)$ is a negligible function.

Figure 13 contains the definition of the security game for a consistent integrity-ensuring measuring scheme ($\mathcal{CMP}$).

Shared by $\mathcal{A}$ and C: random access machine RAM = (M, CPU), program W, integrity ensuring measurement function $\mathcal{M}$P (e.g., an HMAC), security parameter λ, and consistent integrity-ensuring measurement function $\mathcal{CMP}$.

(1) $\mathcal{A}$ chooses the following and provides them to C:
   - Inputs: $x = x_1 || \ldots || x_i$ for RAM.
   - Values: ρ_{init}, ρ_{insert} and ρ_{attest}, all polynomial in λ.
   - Code v to be injected into MEM, and memory location i to insert it (and optionally a list of other memory locations v should be moved to at subsequent rounds after insertion at ρ_{insert}).
(2) C runs Gen(λ) to generate setup parameters.
(3) C simulates ρ_{init} rounds of execution. If round ρ_{insert} is reached, v is inserted into MEM at the beginning of that round. If program halts, go to step 4.
(4) C initiates $\mathcal{CMP}$ by generating a challenge c by invoking Challenge and writing it to the input register. C invokes ROM which contain executable code of $\mathcal{M}$P. C simulates execution of ρ_{attest} rounds. If round ρ_{insert} is reached, v is inserted into MEM at the beginning of that round. If program halts, proceed to step 5.
(5) C interprets data in output register as r, a response to its challenge, and outputs bit b, which is the result of Verify(c, r, $\mathcal{K}$).

Figure 13: $\mathcal{CMP}$ Security Game

B SECURITY ARGUMENTS & CONSIDERATIONS

We consider two approaches: Dec-Lock and All-Lock, and sketch out corresponding security proofs. Security of remaining approaches is quite similar. For the purpose of this section, our instantiations of Dec-Lock and All-Lock is within the HYDRA architecture. Proof sketches are only valid for these specific instantiations since they rely on features ensured by HYDRA. The required (memory isolation and access control) features are instantiated in HYDRA using seL4 which is formally verified. HYDRA uses a secure HMAC as the $\mathcal{M}$P.

B.1 Preliminaries and Assumptions

We capture HYDRA features by the following assumptions:

(1) Assumption-1 (memory access control): memory regions locked, or configured as read-only, cannot be written to by any process.
(2) Assumption-2 (memory isolation): each process, except the attestation one, can only access its own memory space.
(3) Assumption-3 ($\mathcal{M}$P is secure): A secure HMAC is used to implement $\mathcal{M}$P.

B.2 Proof Sketch for Dec-Lock

Considering the security game in Figure 13, there are two cases:

(1) $\mathcal{A}$ supplied $\rho_{insert} \leq \rho_{attest}$
(2) $\mathcal{A}$ supplied $\rho_{insert} > \rho_{attest}$

The first case is trivial, since there is no memory modification after attestation starts, i.e., temporal consistency follows by construction of the case. If everything works as expected, $\mathcal{M}$P computes r on MEM and Verify$(c, r, \mathcal{K})$ should fail, i.e., $b = 0$. b would be 0 because v is now in MEM before $\mathcal{M}$P starts. Thus, the value of r will indicate that; otherwise, $\mathcal{M}$P is insecure, which contradicts Assumption-3. Computation, intermediate and final results of $\mathcal{M}$P cannot be directly affected, since this would violate Assumption-2.

The second case is more subtle. Recall that, in Dec-Lock, entire memory is locked at $t_{vs} = t_{pr} = t_{cs} = \rho_{init}$, and incrementally unlocked as computation of $\mathcal{M}$P proceeds. Assume that memory location i is unlocked after it is processed in round $\rho_{attest} + j$, i.e., one memory location is processed per round after attestation starts. If memory location i, where v is to be inserted, is still locked during ρ_{insert}, i.e., if $\rho_{attest} < \rho_{insert} < \rho_{attest} + j$, then based on Assumption-1 above, v cannot be inserted into MEM. In order to insert v, memory location i has to be unlocked during ρ_{insert}, i.e., $\rho_{attest} + j < \rho_{insert}$; this means that during computation of $\mathcal{M}$P the memory was consistent. Note that the case of $\rho_{attest} + j < \rho_{insert}$ is reduced to case 1 in the next attestation round request. Thus, security follows as the first case above.

B.3 Proof Sketch for All-Lock

Considering the security game in Figure 13, there are two cases:

(1) $\mathcal{A}$ supplied $\rho_{insert} \leq \rho_{attest}$
(2) $\mathcal{A}$ supplied $\rho_{insert} > \rho_{attest}$

The first case is the same as in Dec-Lock.

In the second case, since $\rho_{insert} > \rho_{attest}$ and, at ρ_{attest}, all memory is locked, by Assumption-1 insertion of v into location i will fail, MEM will remain consistent and a correct r will be produced; Verify$(c, r, \mathcal{K})$ will succeed and produce $b = 1$.

C SAMPLE CODE FOR TH$_{\text{FAULT}}$

```
void handle_fault(seL4_CPtr fault_ep, seL4_CPtr main_ep)
{
  seL4_Word sender_badge = 0;
  while(1) {
    seL4_MessageInfo_t tag = seL4_Recv(fault_ep, &sender_badge);
    seL4_Word fault_addr = seL4_GetMR(seL4_VMFault_Addr);
    if(seL4_MessageInfo_get_label(tag) == seL4_Fault_VMFault && !
           sel4utils_is_read_fault() && is_being_attested(fault_addr))
    {
      /* Return back to the main thread with a process causing inconsistency */
      seL4_SetMR(0, sender_badge);
      seL4_Send(main_ep, tag);
    }
  }
}

void create_fault_handler_thread(seL4_CPtr fault_ep, seL4_CPtr main_ep)
{
  sel4utils_thread_t fault_thread;
  seL4_CPtr cspace_cap = simple_get_cnode(&simple);
  int error = sel4utils_configure_thread(&vka, &vspace, &vspace, seL4_CapNull,
         seL4_MaxPrio, cspace_cap, seL4_NilData, &fault_thread);
  assert(error == 0);
  error = sel4utils_start_thread(&fault_thread, handle_fault, (void*) fault_ep,
         (void*) main_ep, 1);
  assert(error == 0);
}
```

D MIGRATORY MALWARE ATTACK

Figure 14 illustrates an example of migratory malware that violates temporal consistency during execution of $\mathcal{M}$P. The attack timeline is as follows:

- At time t_0, malware enters and infects $\mathcal{P}$rv. We assume that malware resides at the tail end of program memory. [8]
- At time $t_1 > t_0$, malware intercepts $\mathcal{V}$rf's attestation request, e.g., by modifying the interrupt handler for the network device driver. It then sets an interrupt timer for t_2 and invokes $\mathcal{M}$P.
- $\mathcal{M}$P runs without interruption from t_1 to t_2.
- At $t_2 > t_1$, malware interrupts $\mathcal{M}$P. It then copies itself to the part of memory that was already measured, erases itself from its prior location, and resumes execution of $\mathcal{M}$P.
- At time t_4, $\mathcal{M}$P completes and produces the measurement for delivery to $\mathcal{V}$rf.

Throughout this process ($t_1 \rightarrow t_4$) malware is never covered by $\mathcal{M}$P. It thus successfully escapes detection, since the measurement reflects a malware-free state.

E IS MIGRATORY MALWARE REALISTIC?

The stance taken in this paper is proactive in nature. One of the goals is a technique that prevents migratory malware from escaping detection (i.e., subverting attestation) on low-end embedded systems. Thus far, there have been no public reports of migratory malware. Nonetheless, we believe that it is realistic and not far-fetched, especially, on low-end embedded systems that involve

[8] If program memory is insufficient to contain both existing firmware and malware, the latter can use the executable compression technique [40] to reduce the sizes of both firmware and itself.

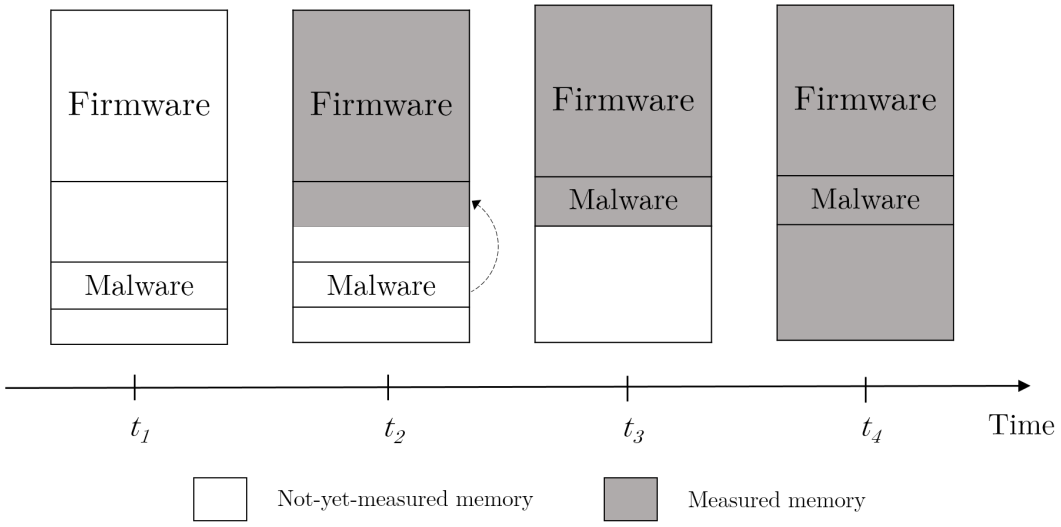


Figure 14: Program memory of infected $\mathcal{P}$rv before (at t_1), during (at t_2, t_3) and after (at t_4) the measurement process.

applications running on “bare metal" and even those capable of supporting a rudimentary microkernel.

In a more traditional computing setting (e.g., PCs, laptops, tablets, and smartphones) anticipated migratory malware resembles the behavior of stealthy viruses [30] that employ various evasion techniques to conceal their existence during a virus scan. Typical evasion techniques involve an operating system and rely on interception of system calls as well as manipulation of returned data. For example, [13] conceals the size of infected files by returning the original size when the DIR command is invoked. Another example is [12, 14, 24] that redirect all access to an infected file to an area storing the original file.

In principle, stealthy malware might also hide its presence by moving itself into an area that has already been covered by a virus scanner, similar to our migratory malware. We believe that this is quite plausible in embedded systems, where a memory migration cannot be detected in software, without using some kind of a temporal consistency mechanism.