UNIVERSITY OF CALIFORNIA,
IRVINE

Secure Remote Attestation for Safety-Critical Embedded and IoT Devices

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Norrathep Rattanavipanon

Dissertation Committee:
Professor Gene Tsudik, Chair
Professor Ardalan Amiri Sani
Professor Ahmad-Reza Sadeghi

2019

DEDICATION

To my family. What a journey.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

It would be impossible for me to complete my Ph.D. journey alone, without any help and support from a lot of people.

First and foremost, I would like to extend my deepest gratitude to my advisor, Prof. Gene Tsudik, for his encouragement, guidance, and support throughout the duration of my Ph.D. years. Without him, I would not even know how to conduct proper research to begin with. I could not imagine having a better advisor for my Ph.D. study.

I would also like to thank the current and previous members of the SPROUT group for fruitful and enjoyable discussion as well as fun activities we had together during my years at UCI: Sky Faber, Ekin Oguz, Cesar Ghali, Tyler Kaczmarek, Christopher Wood, Luca Ferretti, Karim Eldefrawy, Xavier Carpent, Tatiana Bradley, Ivan De Oliveira Nunes, Ercan Ozturk, Yoshimichi Nakatsuka, Michael Stiener, Pier Paolo Tricomi, Seo Yeon Hwang, Sashidar Jakkamsetti, Andrew Searles, Ai Enkoji, Esmerald Aliaj. Special thanks go to Xavier Carpent and Karim Eldefrawy for mentoring me during my first few years at UCI, and also to Ivan De Oliveira Nunes for his tramondous help in my later years.

I would also like to express my gratitude to my defense committee, Ardalan Amiri Sani, Gene Tsudik and Ahmad Reza-Sadeghi, for their valuable time and comments. Without them, I would not be able to complete the degree.

Most importantly, I am extremely grateful for the support and love I have received from my friends and family in Thailand. Because of them, I can fully focus on my Ph.D. study. I am looking forward to starting a new journey with them soon.

This dissertation is the culmination of research conducted at University of California, Irvine. During that time, I was supported by the Royal Thai Government Scholarship and the Graduate Dean's Dissertation Fellowship.

CURRICULUM VITAE

Norrathep Rattanavipanon

EDUCATION

Ph.D. in Computer Science **2019**
University of California, Irvine *Irvine, California*

M.S. in Computer Science **2015**
University of California, Irvine *Irvine, California*

B.S. in Computational Sciences **2013**
University of Michigan – Ann Arbor *Ann Arbor, Michigan*

RESEARCH EXPERIENCE

Graduate Research Assistant **2015–2019**
University of California, Irvine *Irvine, California*

Summer Security Research Assistant **Spring, Summer 2018**
SRI International *Palo Alto, California*

Summer Research Assistant **Summer 2017**
Telefonica *Barcelona, Spain*

TEACHING EXPERIENCE

TA for Computer and Network Security (ICS 134) **Fall 2016**
University of California, Irvine *Irvine, California*

TA for Network and Distributed System Security (ICS 203) **Winter 2017**
University of California, Irvine *Irvine, California*

PROFESSIONAL EXPERIENCE

Software Development Engineer Intern **Summer 2014**
Amazon *Seattle, Washington*

PAPERS IN SUBMISSION OR UNDER REVIEW

A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise **2020**
29th USENIX Security Symposium

Towards Automated Augmentation and Instrumentation for Legacy Cryptographic Executables **2020**
18th International Conference on Applied Cryptography and Network Security

Evil Be Gone: Tackling the TOCTOU Problem in Remote Attestation with RATA **2020**
57th ACM/ESDA/IEEE Design Automation Conference

REFEREED JOURNAL PUBLICATIONS

Remote Attestation via Self-Measurement **2018**
ACM Transactions on Design Automation of Electronic Systems

ASSURED: Architecture for secure software update of realistic embedded devices **2018**
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

REFEREED CONFERENCE PUBLICATIONS

VRASED: A Verified Hardware/Software Co-Design for Remote Attestation **2019**
28th USENIX Security Symposium

Towards Systematic Design of Collective Remote Attestation Protocols **2019**
39th IEEE International Conference on Distributed Computing Systems

Invited: Reconciling remote attestation and safety-critical operation on simple IoT devices **2018**
55th ACM/ESDA/IEEE Design Automation Conference

Temporal consistency of integrity-ensuring computations and applications to embedded systems security **2018**
13th ACM on Asia Conference on Computer and Communications Security

Remote attestation of iot devices via smarm: Shuffled measurements against roving malware **2018**
2018 IEEE International Symposium on Hardware Oriented Security and Trust

ERASMUS: Efficient remote attestation via self-measurement for unattended settings **2018**
Design, Automation & Test in Europe Conference & Exhibition

HYDRA: hybrid design for remote attestation (using a formally verified microkernel) **2017**
10th ACM Conference on Security and Privacy in wireless and Mobile Networks

Fusing hybrid remote attestation with a formally verified microkernel: Lessons learned **2017**
47th IEEE/IFIP International Conference on Dependable Systems and Networks Workshops

Lightweight swarm attestation: a tale of two lisa-s **2017**
12th ACM on Asia Conference on Computer and Communications Security

ABSTRACT OF THE DISSERTATION

Secure Remote Attestation for Safety-Critical Embedded and IoT Devices

By

Norrathep Rattanavipanon

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Professor Gene Tsudik, Chair

In recent years, embedded and cyber-physical systems (CPS), under the guise of Internet-of-Things (IoT), have entered many aspects of daily life. Despite many benefits, this development also greatly expands the so-called attack surface and turns these newly computerized gadgets into attractive attack targets. One key component in securing IoT devices is malware detection, which is typically attained with (secure) remote attestation. Remote attestation is a distinct security service that allows a trusted verifier to verify the internal state of a remote untrusted device. Remote attestation is especially relevant for low/medium-end embedded devices that are incapable of protecting themselves against malware infection. As safety-critical IoT devices become commonplace, it is crucial for remote attestation not to interfere with the device's normal operations. In this dissertation, we identify major issues in reconciling remote attestation and safety-critical application needs. We show that existing attestation techniques require devices to perform uninterruptible (atomic) operations during attestation. Such operations can be time-consuming and thus may be harmful to the device's safety-critical functionality. On the other hand, simply relaxing security requirements of remote attestation can lead to other vulnerabilities. To resolve this conflict, this dissertation presents the design, implementation, and evaluation of several mitigation techniques. In particular, we propose two light-weight techniques capable of providing interruptible attestation modality. In contrast to traditional techniques, our proposed techniques allow interrupts to occur during attestation while ensuring malware detection via shuffled memory traversals or memory locking mechanisms. Another type of techniques pursued in this dissertation aims to minimize the real-time computation overhead during attestation. We propose using periodic self-measurements to measure and record the device's state, resulting in more flexible scheduling of the attestation process and also in no real-time burden as part of its interaction with verifier. This technique is particularly suitable for swarm settings with a potentially large number of safety-critical devices. Finally, we develop a remote attestation HYDRA architecture, based on a formally verified component, and use it as a building block in our proposed mitigation techniques. We believe that this architecture may be of independent interest.

Chapter 1

Introduction

In recent years, the number and variety of special-purpose computing devices have increased dramatically. This includes all kinds of embedded devices, cyber-physical systems (CPS) and Internet-of-Things (IoT) gadgets. Such devices are increasingly deployed in various smart settings, such as homes, offices, factories, automotive and public venues. Despite their various benefits, these devices unfortunately also represent natural and attractive attack targets.

In the context of actuation-capable devices, malware can impact both security and physical safety, e.g., as demonstrated by Stuxnet [87] attack on centrifuges in an Iranian nuclear facility. Whereas, for sensing devices, malware can undermine privacy by obtaining ambient information about nearby activities. Furthermore, clever malware can attack availability of IoT devices by turning them into zombies that can become sources for distributed denial-of-service (DDoS) attacks. For example, in Fall 2016, a multitude of compromised smart cameras and DVRs formed the Mirai Botnet [4] which was used to mount a massive-scale DDoS attack.

Unfortunately, security is typically not the highest priority for IoT 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 detecting the presence of malware. This typically requires some form of *Remote Attestation* ($\mathcal{R}\mathsf{A}$) – a distinct security service for detecting malware on CPS, embedded and IoT devices. $\mathcal{R}\mathsf{A}$ is especially applicable to 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, e.g., anti-virus and/or intrusion detection software.

In the past two decades, many $\mathcal{R}\mathsf{A}$ techniques with various assumptions and complexities have been proposed. To the best of our knowledge, none of the prior techniques has investigated and evaluated the potential impact of secure $\mathcal{R}\mathsf{A}$ on practical IoT deployment settings. As IoT devices are often used in safety-critical and real-time operations, it is paramount that a $\mathcal{R}\mathsf{A}$ technique *must not* interfere with normal operations of such devices. Unfortunately, this is mostly not the case. For instance, one widely accepted consensus on a security requirement is atomic and non-interruptible execution of $\mathcal{R}\mathsf{A}$ functionality [31, 33]. However, $\mathcal{R}\mathsf{A}$ execution can be quite time-consuming.

Consider a (sensing-actuating) fire alarm application running on a low/medium-end embedded device, powered by an ODROID-XU4 development board [37]. Under normal operating conditions, this application periodically checks the value of its temperature readings and triggers an alarm whenever the value exceeds a certain threshold. As mentioned earlier, to be secure, the device must execute the $\mathcal{R}\mathsf{A}$ functionality atomically. Attesting this device with 1GB RAM requires approximately 7 seconds. However, if a fire breaks out soon after execution starts, it would take a relatively long time for the application to regain control, sense the fire, and respond appropriately. It is apparent that enforcing this security requirement can be harmful since precious time lost might cause disastrous consequences. Resolving this conflict is the challenge that we explore in this dissertation.

1.1 Contribution

The goal of this dissertation is to investigate $\mathcal{R}\mathsf{A}$ in safety-critical IoT settings. To this end, this dissertation makes the following contributions:

- We identify issues in existing $\mathcal{R}\mathsf{A}$ techniques that arise in safety-critical applications. To be secure, existing techniques enforce the devices to perform uninterruptible (atomic) operations during attestation. We demonstrate that such operations can be time-consuming in practice. As a result, $\mathcal{R}\mathsf{A}$ functionality can directly interfere with, and thus be harmful to, device's safety-critical functionality and general availability. On the other hand, we also show that simply relaxing security requirements is non-trivial and can result in vulnerabilities.

- We explore the solution landscape of reconciling requirements of safety-critical operation with those of secure $\mathcal{R}\mathsf{A}$. We propose three distinct solutions, each making different assumptions about the underlying hardware device as well as providing different degrees of availability for the device. Our three solutions are mutually exclusive and, in principle, can be used in conjunction with one another to ensure higher availability of safety-critical devices.

- We also study the problem of secure $\mathcal{R}\mathsf{A}$ and safety-critical operation in large-scale settings, where $\mathcal{R}\mathsf{A}$ is performed over a potentially large network of IoT devices. We propose two practical attestation schemes with different communication and computation complexities, and analyze their applicability to safety-critical settings.

- Throughout the process of developing these solutions, we also design an independent $\mathcal{R}\mathsf{A}$ architecture and use it as a building block in our proposed solutions. Contrary to all previous relevant efforts, our design requires no modifications to the underlying hardware, which makes it deployable on off-the-shelf IoT devices. We demonstrate such feasibility and practicality by implementing full prototypes of our design on two commercially available development boards.

1.2 Outline

We begin by surveying the current landscape of $\mathcal{R}\mathsf{A}$ techniques in Chapter 2. In Chapter 3, we motivate our work by showing how existing $\mathcal{R}\mathsf{A}$ techniques fail to satisfy operational requirements of safety-critical IoT devices, and then identify the tussle between the security requirements and safety-critical application needs. The overview of our solutions is presented at the end of the same chapter. Chapter 4 presents a new $\mathcal{R}\mathsf{A}$ architecture that serves as a building block in our proposed solutions. Our approaches are described in Chapters 5, 6, and 7. In Chapter 8, we investigate issues that arise in settings with groups of devices. We conclude this dissertation in Chapter 9.

Chapter 2

Related Work

This chapter overviews the current research landscape in $\mathcal{R}\mathsf{A}$. $\mathcal{R}\mathsf{A}$ is a security service that can aid in malware detection; it is typically realized as an interactive protocol between a trusted verifier ($\mathcal{V}\mathsf{rf}$) and a potentially compromised remote prover device ($\mathcal{P}\mathsf{rv}$). As shown in Figure 2.1, current $\mathcal{R}\mathsf{A}$ techniques can be clustered into four major categories, based on: their requirements about the underlying architecture of $\mathcal{P}\mathsf{rv}$ (Section 2.1), detection guarantees (Section 2.2), assumptions about network communication (Section 2.3), and the adversarial model (Section 2.4). Using the proposed taxonomy, we compare current techniques (Section 2.5) and discuss the scope of this dissertation (Section 2.6).

2.1 Architectural Requirements

The first category in our taxonomy provides a qualitative measure of architectural requirements for a given $\mathcal{R}\mathsf{A}$ technique to operate securely. It categorizes $\mathcal{R}\mathsf{A}$ techniques based on the amount of hardware necessary to provide all $\mathcal{R}\mathsf{A}$ *security* guarantees.

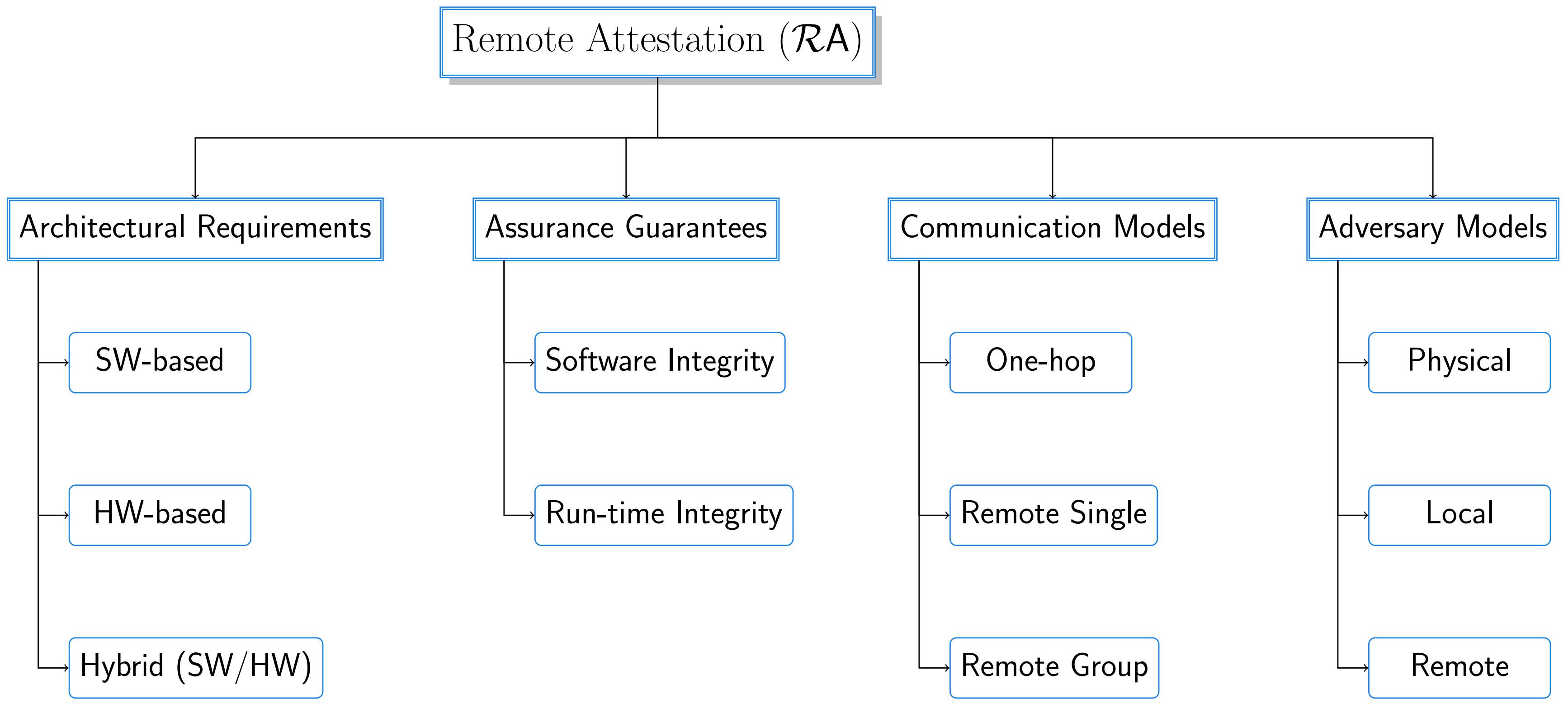


Figure 2.1: $\mathcal{R}$A Taxonomy.

2.1.1 Hardware-based Techniques

We consider an $\mathcal{R}\mathsf{A}$ technique to be hardware-based if it requires either (1) $\mathcal{R}\mathsf{A}$ functionality to be housed entirely within dedicated hardware (e.g., co-processors), or (2) modifications or additions to the underlying instruction set architecture. We now overview three important main types of hardware-based technique.

Trusted Platform Module (TPM) [86]. A TPM is an international standard for a secure co-processor designed to protect cryptographic keys, and utilize them to encrypt or digitally sign data. At boot time, a TPM computes a hash of loaded software and stores the result in Platform Configuration Registers (PCR). $\mathcal{R}\mathsf{A}$ functionality is then provided by having the TPM sign these values with an attestation key along with a random challenge, provided by $\mathcal{V}\mathsf{rf}$, and submit the computed result to $\mathcal{V}\mathsf{rf}$. Security of TPM-based $\mathcal{R}\mathsf{A}$ relies on the fact all of its key components (including PCR, the attestation key and the hash engine) are implemented entirely in hardware, and thus only accessible to the TPM itself. Subsequent work [71] proposed an emulation of a TPM by leveraging the existing trusted execution environment ARM TrustZone [6].

Intel SGX [23]. In 2013, Intel introduced a new set of CPU instructions, termed Software Guard Extensions (Intel SGX). SGX enables realizing an isolated execution environment (or *enclave*). Using hardware mechanisms, SGX guarantees that: (1) contents (i.e., code and data) inside each enclave are inaccessible by any process except the enclave itself, (2) execution of enclave's code must start at a dedicated entry point, and (3) CPU executions (e.g., faults and interrupts) inside an enclave are handled securely without leaking its private information. Based on these properties, Intel claims to ensure confidentiality and integrity of code and data running inside an enclave. To perform $\mathcal{R}\mathsf{A}$, an enclave first proves its identity to a privileged *quoting enclave* via local attestation – a hardware-based mechanism used by an enclave to convince the other enclave that they are running on the same SGX-enabled platform. After successfully asserting the identity, the quoting enclave creates an attestation report by signing a hash of the application enclave's contents. The attestation key is initially retrieved and protected by another privileged *provisioning enclave*. This report can then be used as evidence to convince $\mathcal{V}\mathsf{rf}$ that the application enclave is running the expected software.

Sancus [63]. Sancus proposes a security architecture for embedded devices without a trusted codebase. Similar to SGX, Sancus provides a means of $\mathcal{R}\mathsf{A}$ based on isolation between protected software modules. In Sancus, isolation is achieved by additional CPU instructions enforcing the following: (1) code of all protected modules is immutable, (2) data of a given software module is only accessible while the code of the same module is being executed, and (3) code can only be executed by jumping to a well-defined entry point. In the context of $\mathcal{R}\mathsf{A}$, these properties guarantee confidentiality of the attestation key residing in a protected module when such a module requests an attestation report via a special CPU instruction, `MAC-Seal`. Upon completing execution of the `MAC-Seal` instruction, the attestation report becomes available to the requesting module; it then can be used to prove to $\mathcal{V}\mathsf{rf}$ as well as other software modules that the module is correctly loaded and running on the device.

2.1.2 Software-based Techniques

On the other end of the spectrum, software-based techniques require no hardware support at all, and perform $\mathcal{R}\mathsf{A}$ using a custom function implemented entirely in software. This approach relies on precise timing of a $\mathcal{R}\mathsf{A}$ protocol and/or memory constraints on $\mathcal{P}\mathsf{rv}$'s device.

Precise Timing. *Pioneer* [79] is a prominent example of this approach. It constructs a one-time special checksum function that covers memory in an unpredictable (rather than contiguous) fashion. Traversing memory in an unpredictable order prevents a "memory shadowing" attack [21] where an adversary redirects attesting memory locations to different locations storing the correct values. Also, any attempt to forge a checksum output results in a noticeable delay which will be detected by $\mathcal{V}\mathsf{rf}$. A similar approach has been adopted by other techniques targeting specific deployment settings, e.g., *Viper* [52] that uses a time-sensitive checksum computation to verify integrity of peripherals' firmware, while *SWATT* [80] uses a similar method to verify memory of an embedded device. In all software-based methods, $\mathcal{V}\mathsf{rf}$ is convinced that prover's software is in a benign state only if the checksum is correct (i.e., corresponds to correct software) and is received within a (predetermined) time limit. Due to strict timing constraints during verification, the implementation of the checksum function needs to be time-optimal; otherwise, an adversary can easily forge an attestation result by simply using a faster implementation of the same checksum function on a copy of valid software [21]. For the same reason, communication delay between $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$ must also be fixed or negligible (compared to the time required to compute checksum).

Memory Constraint. An alternative approach is to exploit memory constraints of $\mathcal{P}\mathsf{rv}$'s device. The idea behind this approach is to not leave any empty memory space for the malicious code to hide in during the checksum computation. Choi et al. [22] propose to fill all unused memory with pseudorandom numbers derived from a $\mathcal{V}\mathsf{rf}$-generated random seed. Afterward, $\mathcal{P}\mathsf{rv}$ can construct an attestation report by computing a hash over its entire memory, and submit it to $\mathcal{V}\mathsf{rf}$. $\mathcal{V}\mathsf{rf}$ can validate the received report since it knows the expected entire memory snapshot, which consists of original memory contents plus contents of previously empty memory filled with $\mathcal{V}\mathsf{rf}$-chosen pseudorandom numbers. Yang et al. [91] further extended this approach to perform distributed software-based attestation, where integrity of $\mathcal{P}\mathsf{rv}$'s software is determined in a distributed manner by all $\mathcal{P}\mathsf{rv}$'s neighbors. Security of this approach relies on two assumptions. First, it assumes space optimality of original memory contents (including the checksum code), i.e., they should be uncompressible; otherwise, an adversary can simply compress and gain enough free space not filled by pseudorandom numbers to store and run malicious code to evade attestation [21]. This assumption can be dropped if $\mathcal{P}\mathsf{rv}$ has a read-only memory region to store and execute attestation code, as suggested in [69]. The second assumption is that $\mathcal{V}\mathsf{rf}$ must be aware of all communication between $\mathcal{P}\mathsf{rv}$ and other entities. This assumption is necessary to prevent a so-called "proxy" attack where malicious $\mathcal{P}\mathsf{rv}$ asks for help from a more powerful accomplice device to compute a valid response.

A more recent work [36] leverages both precise timing and memory constraint characteristics in order to to construct software-based attestation for verifying integrity of peripheral's software at boot-time. In particular, the authors in [36] propose a new primitive, called *randomized polynomial*, and mathematically proves its space-time optimality in a formal model of their target $\mathcal{P}\mathsf{rv}$-s. Intuitively, time optimality guarantees that it is impossible for malware to evaluate a randomized polynomial faster than the theoretical lower bound; whereas space optimality implies that no malware can hide in code implementing a randomized polynomial. $\mathcal{P}\mathsf{rv}$ creates an attestation response by evaluating the randomized polynomial on its own memory and along with a $\mathcal{V}\mathsf{rf}$-generated challenge. $\mathcal{V}\mathsf{rf}$ then concludes that $\mathcal{P}\mathsf{rv}$ is in a malware-free state if it receives the correct response within the theoretical lower bound.

2.1.3 Hybrid (HW/SW) Techniques

Hybrid techniques use hardware and software co-design to provide security guarantees for $\mathcal{R}\mathsf{A}$. In this dissertation, we consider an $\mathcal{R}\mathsf{A}$ technique to be hybrid if its hardware changes is simple as well as does not include any changes in the underlying instruction set architecture. Examples of common hardware components include a simple read-only memory storage (ROM), or dedicated hardware to monitor and enforce memory access control rules. Below, we overview two important $\mathcal{R}\mathsf{A}$ architectures that have become a foundation for subsequent work in hybrid $\mathcal{R}\mathsf{A}$.

SMART(+) [31, 12]. SMART is the first hybrid technique for $\mathcal{R}\mathsf{A}$ with (allegedly) minimal hardware modifications to current micro-controller units (MCUs). It stipulates that attestation software and attestation key reside in immutable storage (e.g., ROM) and are guarded by MCU access control rules. The latters are implemented by modifying the MCU's data bus, and enforce the following: (1) the key is only accessible to attestation software, and (2) execution of attestation code is atomic, i.e., only starts from its first instruction and finishes at the last instruction. Whenever any violation of these rules occurs, SMART erases all data memory and resets the device. SMART's attestation software guarantees the non-interruptible execution of attestation functionality, which is necessary to prevent code-reuse attacks [74]. Also, SMART performs static analysis on attestation software to ensure no key leak occurs through CPU registers after its execution. All of these properties are further systematically analyzed in [33] and argued to be necessary and sufficient for providing secure $\mathcal{R}\mathsf{A}$. Subsequent work in [12] extended SMART to defend against denial-of-service (DoS) attacks that try to impersonate $\mathcal{V}\mathsf{rf}$. This extended variant (referred to as SMART+) additionally requires $\mathcal{P}\mathsf{rv}$ to have a Reliable Read-Only Clock (RROC), needed to perform $\mathcal{V}\mathsf{rf}$ authentication and prevent replay, reorder and delay attacks. To ensure reliability, RROC must not be modifiable by software. In SMART+, upon receiving a $\mathcal{V}\mathsf{rf}$ request, ROM-resident attestation code checks the request's freshness using RROC, authenticates it, and only then proceeds to perform $\mathcal{R}\mathsf{A}$.

TrustLite [47]. TrustLite extends SMART by supporting strong isolation of software modules, called trustlets. Such isolation is guaranteed by the use of an *execution-aware memory protection unit* (EA-MPU), which can be programmed at compile-time. TrustLite can realize $\mathcal{R}\mathsf{A}$ by configuring its EA-MPU rules in such a way that: (1) no other trustlets can access data (which also includes the attestation key) inside the attestation trustlet, and (2) execution of the attestation trustlet can only start from the well-defined entry point. TrustLite also supports secure interrupt handling by modifying the CPU Exception Engine to store the trustlet's context in a protected region and clear the CPU registers before switching to an untrusted interrupt handler. In principle, this feature can be used to prevent the key leakage through CPU registers when the attestation trustlet is interrupted by untrusted code. Nonetheless, we demonstrate in Chapter 6 that secure interrupt handling, along with the aforementioned EA-MPU rules, is insufficient to provide $\mathcal{R}\mathsf{A}$ in TrustLite. One also needs some kind of temporal consistency mechanism, e.g., disabled interrupts or memory locking, in order to prevent malicious code from evading detection by issuing an interrupt and relocating itself to an already-attested memory region. A follow-on effort, called TyTAN [11], adopted a similar approach while providing additional real-time guarantees. TyTAN's key feature is to allow EA-MPU rules to be programmable at runtime. Therefore, trustlets in TyTAN can be dynamically loaded and unloaded on demand at runtime, which was not previously possible in TrustLite.

2.2 Assurance Guarantees

This category refers to a type of assurance guarantees obtained by $\mathcal{V}\mathsf{rf}$ about $\mathcal{P}\mathsf{rv}$ after receiving a correct attestation response from $\mathcal{P}\mathsf{rv}$. We classify such assurance into two groups: software integrity and run-time integrity.

2.2.1 Software Integrity

The majority of existing work in $\mathcal{R}\mathsf{A}$, including the schemes in Section 2.1, focuses on detecting malware presence by verifying integrity of $\mathcal{P}\mathsf{rv}$'s software. This detection guarantee is typically achieved by a challenge-response protocol, where $\mathcal{P}\mathsf{rv}$ first receives a challenge from $\mathcal{V}\mathsf{rf}$, and generates a response by computing an integrity check (e.g., checksum or keyed-hash) of the received challenge over its own memory. $\mathcal{V}\mathsf{rf}$ could also pre-compute all safe/legal values ahead of time. Since the verification process assumes $\mathcal{V}\mathsf{rf}$ knows all valid memory states, previous work usually targets attestation of static memory regions (i.e., code) while ignoring highly variable memory regions (i.e., data). In Chapter 6, we propose a simple technique to achieve integrity of *both* code and data on $\mathcal{P}\mathsf{rv}$.

In this detection goal, the correct response assures that $\mathcal{P}\mathsf{rv}$'s software is in a legitimate state *during* $\mathcal{R}\mathsf{A}$. However, it does not provide any guarantees about the history of $\mathcal{P}\mathsf{rv}$'s software states, e.g., before performing $\mathcal{R}\mathsf{A}$. In Chapter 7, we address this issue by presenting an approach based on *periodic self-measurements* that allows $\mathcal{P}\mathsf{rv}$ to securely and periodically record its software state. These recordings can then be used as evidence to convince $\mathcal{V}\mathsf{rf}$ that $\mathcal{P}\mathsf{rv}$ has been in a healthy state in between two $\mathcal{R}\mathsf{A}$ sessions. This idea was pursued by Ibrahim et al. [39] in parallel to our work. Nonetheless, our work shows that the approach based on periodic self-measurements can further remedy the conflict between $\mathcal{R}\mathsf{A}$ and safety-critical application needs, and thus is better suited for safety-critical applications – the primary goal of this dissertation.

2.2.2 Run-time Integrity

Instead of verifying memory content, this type of assurance guarantee aims to detect malware presence during execution of target software by verifying its run-time properties. It is mainly motivated by the fact that software integrity is not enough to guarantee that execution of attested software will ever happen. Malware may (re-)infect $\mathcal{P}\mathsf{rv}$ after completing $\mathcal{R}\mathsf{A}$ and before executing attested software. This is referred to as the time-of-check to time-of-use (TOCTOU) problem [13].

CFLAT [1] is among the first to propose a solution for remotely verifying $\mathcal{P}\mathsf{rv}$'s execution paths. It leverages a trusted execution environment (e.g., TrustZone [6]) to securely compute a hash of the exact sequence of executed instructions on $\mathcal{P}\mathsf{rv}$'s device. The result allows $\mathcal{V}\mathsf{rf}$ to determine whether a particular code is executed on $\mathcal{P}\mathsf{rv}$, as well as whether its execution is not tampered with and strictly follows one of the valid control-flow paths. C-FLAT imposes significant run-time overhead on $\mathcal{P}\mathsf{rv}$ as it requires performing multiple context switches between secure and non-secure environments during $\mathcal{R}\mathsf{A}$. A follow-on work, called LoFAT [25], aims to overcome this limitation by implementing an $\mathcal{R}\mathsf{A}$ engine entirely in hardware. A more recent work, LiteHAX [24], incorporates additional data-flow information into its $\mathcal{R}\mathsf{A}$ scheme in order to prevent data-oriented attacks.

2.3 Communication Models

$\mathcal{R}\mathsf{A}$ techniques make various assumptions about the underlying network communication model between $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$. We divide them into three settings: one-hop settings, remote settings, and remote group settings.

2.3.1 One-hop Setting

In this setting, $\mathcal{P}\mathsf{rv}$ is located exactly one-hop away from $\mathcal{V}\mathsf{rf}$. This means that $\mathcal{V}\mathsf{rf}$ is capable of directly communicating with $\mathcal{P}\mathsf{rv}$ without requiring any intermediate nodes. Examples of this model include communication between peripherals and their host CPU via a Peripheral Component Interconnect (PCI) Bus or intra-vehicular connectivity utilizing a Controller Area Network (CAN) Bus network. The one-hop communication model is commonly used to capture one of the following network characteristics: (1) a reliable network where no packet loss occurs, i.e., communication between $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$ is lossless, or (2) a local area network where $\mathcal{V}\mathsf{rf}$ overhears all $\mathcal{P}\mathsf{rv}$'s incoming and outgoing communication. We also note that security of software-based $\mathcal{R}\mathsf{A}$ techniques (See Section 2.1.2) makes the same assumption, limiting their applicability to only the one-hop communication setting.

2.3.2 Remote Setting (with Single $\mathcal{P}\mathsf{rv}$)

This setting allows $\mathcal{V}\mathsf{rf}$ and $\mathcal{P}\mathsf{rv}$ to be remote to each other, i.e., separated by an arbitrary many hops. Examples include the Internet and wireless mesh networks. The assumption about constant/negligible network delays is not realistic in this setting because of packet loss or variable latency. Therefore, $\mathcal{R}\mathsf{A}$ solutions based on software-based techniques are not applicable to this setting.

2.3.3 Remote Group Setting

Here, we consider a remote group setting where $\mathcal{V}\mathsf{rf}$ communicates with a network of interconnected $\mathcal{P}\mathsf{rv}$-s over intermediate hops. The goal for $\mathcal{R}\mathsf{A}$ in this setting is for $\mathcal{V}\mathsf{rf}$ to assert integrity of a group of $\mathcal{P}\mathsf{rv}$-s *as a whole.* $\mathcal{V}\mathsf{rf}$ may also be interested in knowing "which" $\mathcal{P}\mathsf{rv}$-s are infected. In this dissertation, we refer to $\mathcal{R}\mathsf{A}$ techniques in this setting as *collective remote attestation* (cRA). Besides security, efficiency and scalability are also important when designing a cRA scheme.

SEDA [7] is the first $\mathcal{R}\mathsf{A}$ scheme tailored for the group setting. In SEDA, $\mathcal{V}\mathsf{rf}$ starts the cRA protocol by sending a request to an initiator device, selection of which is flexible. Having received a request, a device accepts the sender as its parent and forwards that request to its neighbors. An attestation report of any device is created – and protected using a secret key (distributed as part of the off-line phase) shared between that device and its parent – and sent to its parent device. Once it receives all of its children's reports, a device aggregates them into a single report. This process is repeated until the initiator receives all reports from its children. The initiator then combines these reports, signs them using its private key and returns the final result to $\mathcal{V}\mathsf{rf}$. Security of SEDA relies on the underlying hybrid $\mathcal{R}\mathsf{A}$ architecture such as SMART [31] or TrustLite [47].

SANA [76] extends SEDA by providing a more efficient and scalable cRA scheme based upon Optimistic Aggregate Signatures (OAS). OAS allows many individual signatures to be efficiently combined into a single aggregated signature, which can be verified much faster than all individual signatures. SANA's scalability is demonstrated via simulation showing that it can attest a million devices in 2.5 seconds.

Chapter 8 of this dissertation proposes two practical cRA schemes. Our aim when designing such schemes is to narrow the gap between paper-design techniques, such as SEDA and SANA, and realistic performance assessment and practical deployment. Throughout the design process, we also define a qualitative measure for cRA, i.e., *Quality of Swarm Attestation* (QoSA). QoSA reflects $\mathcal{V}\mathsf{rf}$'s information requirements and allows comparison across cRA techniques.

2.4 Adversarial Models

In the context of $\mathcal{R}\mathsf{A}$, the adversarial models typically fall into three clusters: physical, local and remote adversaries.

2.4.1 Physical Adversary

This type of adversary has full physical access to $\mathcal{P}\mathsf{rv}$. It can launch any hardware-based attacks on the device. This includes mounting hardware-based side-channel attacks, or physically removing any (even protected) $\mathcal{P}\mathsf{rv}$'s memory (e.g., RAM, ROM or flash) and extracting secrets from it. In addition, it can induce hardware faults as well as directly access and modify hardware states of $\mathcal{P}\mathsf{rv}$. In the context of $\mathcal{R}\mathsf{A}$, this type of adversary is considered to be the most difficult to mitigate, and generally excluded from the threat model considered by most $\mathcal{R}\mathsf{A}$ techniques. Nonetheless, a few techniques [38, 48] attempt to detect, and mitigate damages from, physical attacks in a network of devices. For example, *DARPA* [38] proposes a mitigation technique based on the rationale that, an adversary needs to spend a non-negligible amount of time to physically compromise that device. To detect device absence, DARPA requires each device to periodically monitor other devices by recording their *heartbeat*, at regular time intervals. $\mathcal{V}\mathsf{rf}$ can then detect any absent device when collecting these heartbeats. DARPA can also be used in a conjunction with other cRA schemes to provide protection in the presence of a physical adversary.

2.4.2 Local Adversary

We consider an adversary to be *local* if it is located sufficiently near $\mathcal{P}\mathsf{rv}$ to have control over $\mathcal{P}\mathsf{rv}$'s communication. In particular, it can eavesdrop on, and manipulate, i.e., insert, drop, record and replay, any messages between $\mathcal{V}\mathsf{rf}$ and $\mathcal{P}\mathsf{rv}$. This adversarial capability is akin to the Dolev-Yao adversarial model [27], which is widely used to formally prove properties of cryptographic protocols. Software-based $\mathcal{R}\mathsf{A}$ techniques generally do not consider this type of adversary, because their underlying assumptions strongly rely on adversarial control over $\mathcal{P}\mathsf{rv}$'s communication (see Section 2.1.2 for more detail).

2.4.3 Remote Adversary

Remote adversaries exploit vulnerabilities in $\mathcal{P}\mathsf{rv}$'s software to inject malware over the network. By introducing malware onto $\mathcal{P}\mathsf{rv}$, this adversary controls $\mathcal{P}\mathsf{rv}$'s entire software state, including all code and data. Specifically, it can modify any writable memory and read any memory that is not explicitly protected by (hardware-enforced) access control rules, i.e., it can read anything (including secrets) that is not explicitly protected by the trusted hardware. Malware can also attempt to evade detection by erasing or re-locating itself during $\mathcal{R}\mathsf{A}$. All $\mathcal{R}\mathsf{A}$ techniques aim to provide security against remote adversaries.

2.5 Comparison

Table 2.1 summarizes notable $\mathcal{R}\mathsf{A}$ research based on the taxonomy proposed earlier in this chapter. We observe that all software-based techniques consider remote, but not physical and local, attacks in their adversarial model. On the other hand, most hardware-based and hybrid techniques aim to prevent both local and remote adversaries, but not physical adversaries. Other than imposing ubiquitous tamper-resistant hardware, the only practical means of mitigating physical attacks is by heartbeat-based absence detection [38]. Most $\mathcal{R}\mathsf{A}$ techniques consider this to be an orthogonal issue. Furthermore, software-based techniques are suitable only for one-hop $\mathcal{P}\mathsf{rv}$-$\mathcal{V}\mathsf{rf}$. Hardware-based and hybrid techniques can be used in both one-hop and remote settings. Finally, all techniques aiming to provide run-time integrity guarantees are hardware-based.

2.6 Scope

As this dissertation focuses on low/medium-end embedded and IoT devices, we consider hardware-based techniques too expensive. This is because such techniques require dedicated hardware features that are only available in more powerful (high-end) devices, e.g., personal computers, laptops or smartphones. The same features are considered a "luxury" for low/medium-end embedded and IoT devices. On the other hand, software-based techniques rely on strong assumptions about their adversarial capabilities, which are unrealistic in networked (multi-hop) settings. Additionally, it is difficult to prove time- and/or space-optimality of the checksum implementation in practice. In fact, among all proposed software-based techniques, [36] is the only technique capable of successfully proving such properties.

In this dissertation, we consider hybrid techniques to be the best fit for our target devices as

				Architecture (Sec 2.1)			Guarantee (Sec 2.2)		Communication (Sec 2.3)			Adversary (Sec 2.4)		
Authors	Year First Published	Paper Name	Reference	HW-based	SW-based	Hybrid	Software Integrity	Run-time Integrity	One-hop	Remote	Remote Group	Physical	Local	Remote
Seshadri et al.	2004	SWATT	[80]	○	●	○	●	○	●	○	○	○	○	●
Seshadri et al.	2005	Pioneer	[79]	○	●	○	●	○	●	○	○	○	○	●
Choi et al.	2007	-	[22]	○	●	○	●	○	●	○	○	○	○	●
Yang et al.	2007	-	[91]	○	●	○	●	○	●	○	○	○	○	●
TCG	2009	TPM	[86]	●	○	○	●	○	○	●	○	○	●	●
Li et al.	2011	Viper	[52]	○	●	○	●	○	●	○	○	○	○	●
ElDefrawy et al.	2012	SMART	[31]	○	○	●	●	○	○	●	○	○	●	●
Noorman et al.	2013	Sancus	[63]	●	○	○	●	○	○	●	○	○	●	●
Intel	2013	SGX	[23]	●	○	○	●	○	○	●	○	○	●	●
Koeberl et al.	2014	TrustLite	[47]	○	○	●	●	○	○	●	○	○	●	●
Asokan et al.	2015	SEDA	[7]	○	○	●	●	○	○	○	●	○	●	●
Ambrosin et al.	2016	SANA	[76]	○	○	●	●	○	○	○	●	○	●	●
Ibrahim et al.	2016	DARPA	[38]	○	○	●	●	○	○	○	●	●	●	●
Abera et al.	2016	CFLAT	[1]	●	○	○	○	●	○	●	○	○	●	●
Dessouky et al.	2017	LO-FAT	[25]	●	○	○	○	●	○	●	○	○	●	●
Dessouky et al.	2018	LiteHAX	[24]	●	○	○	○	●	○	●	○	○	●	●
Gligor and Woo	2018	-	[36]	○	●	○	●	○	●	○	○	○	○	●

● is/provides/assumes/considers ○ is not/not provide/not assume/not consider

Table 2.1: Summary of $\mathcal{RA}$ research discussed throughout this chapter in a chronological order

they require minimal additional hardware features and are thus suitable for cost-sensitive IoT devices. Consequently, we choose to present our $\mathcal{R}\mathsf{A}$ approaches in Chapters 4, 5, 6, 7 and 8 of this dissertation based on hybrid techniques. We also consider the same adversarial models as those considered by hybrid techniques. That is, we only consider remote and local adversaries while physical adversaries are out of scope in this dissertation. Finally, as the first step to reconcile secure $\mathcal{R}\mathsf{A}$ and safety-critical operation needs, our work only targets software integrity as the primary assurance guarantees expected from $\mathcal{R}\mathsf{A}$. Adapting our solutions to techniques that guarantee run-time integrity is an interesting future direction.

Chapter 3

Remote Attestation in Safety-Critical Settings

In this chapter, we motivate the main problem that we study in this dissertation, i.e., reconciling $\mathcal{R}\mathsf{A}$ with safety-critical applications. We start by giving an overview of $\mathcal{R}\mathsf{A}$ (Section 3.1). We then proceed to explain why existing $\mathcal{R}\mathsf{A}$ techniques (overviewed in the previous chapter) fail to satisfy the needs for safety-critical application. We also show that resolving this conflict is non-trivial; naïvely relaxing security requirements can lead to (additional) vulnerabilities (Section 3.2). Finally, we conclude by presenting the high-level overview of our solutions to this problem (Section 3.3).

3.1 $\mathcal{R}\mathsf{A}$ Overview

$\mathcal{R}\mathsf{A}$ is a security service that facilitates detection of malware presence on a remote device. $\mathcal{R}\mathsf{A}$ is especially applicable to embedded and IoT devices incapable of defending themselves against malware infection. This is in contrast to more powerful devices (both embedded and general-purpose) that can rely on sophisticated anti-malware protection. $\mathcal{R}\mathsf{A}$ involves verification of the current internal state (i.e., RAM and/or flash) of an untrusted remote hardware platform (prover or $\mathcal{P}\mathsf{rv}$) by a trusted entity (verifier or $\mathcal{V}\mathsf{rf}$). If $\mathcal{V}\mathsf{rf}$ detects malware presence, $\mathcal{P}\mathsf{rv}$'s software can be reset or rolled back and out-of-band measures can be taken to prevent similar infections. In general, $\mathcal{R}\mathsf{A}$ can help $\mathcal{V}\mathsf{rf}$ establish a static or dynamic root of trust in $\mathcal{P}\mathsf{rv}$ and can also be used to construct other security services, such as software updates [8], secure memory erasure [69, 66], and secure code execution [67].

3.1.1 $\mathcal{R}\mathsf{A}$ Blueprint

As illustrated in Figure 1, $\mathcal{R}\mathsf{A}$ is typically obtained via a simple challenge-response protocol:

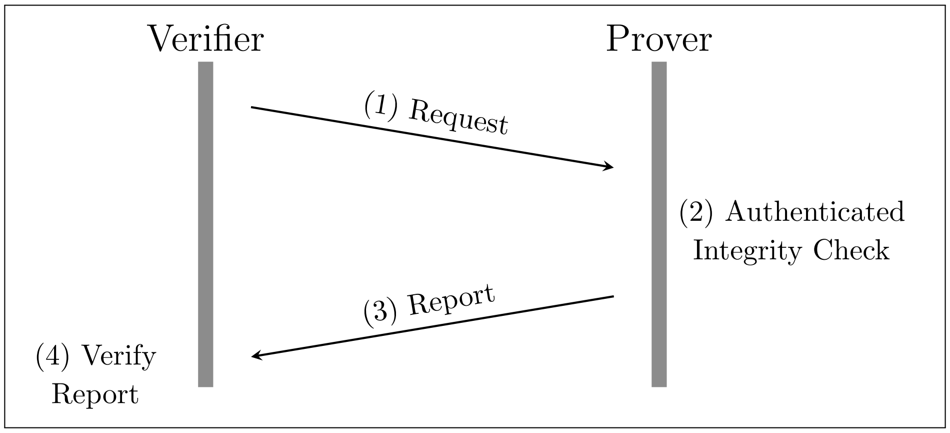


Figure 3.1: Blueprint of a typical $\mathcal{R}\mathsf{A}$ protocol

1. $\mathcal{V}\mathsf{rf}$ sends a challenge-bearing attestation request to $\mathcal{P}\mathsf{rv}$. This request might also contain a token derived from a secret that allows $\mathcal{P}\mathsf{rv}$ to authenticate $\mathcal{V}\mathsf{rf}$.

2. $\mathcal{P}\mathsf{rv}$ receives this request and computes an *authenticated integrity check* over its memory and the supplied challenge. The memory region might be either pre-defined, or explicitly specified in the request. In the latter case, authentication of $\mathcal{V}\mathsf{rf}$ in step (1) is paramount to the overall security, privacy and availability of $\mathcal{P}\mathsf{rv}$, as the request can specify arbitrary memory regions.

3. $\mathcal{P}\mathsf{rv}$ sends the computed attestation report to $\mathcal{V}\mathsf{rf}$.

4. $\mathcal{V}\mathsf{rf}$ receives the result, and verifies whether it corresponds to a valid memory state.

The authenticated integrity check can be realized as a Message Authentication Code (MAC) over $\mathcal{P}\mathsf{rv}$'s memory. However, computing a MAC requires $\mathcal{P}\mathsf{rv}$ to have a unique secret key (denoted by K) shared with $\mathcal{V}\mathsf{rf}$. This key must reside in secure storage, where it is not accessible to any software running on $\mathcal{P}\mathsf{rv}$, except for attestation code. Since most $\mathcal{R}\mathsf{A}$ threat models assume a fully compromised software state on $\mathcal{P}\mathsf{rv}$, secure storage implies some level of hardware support.

3.1.2 Coverage of $\mathcal{R}\mathsf{A}$

The usual $\mathcal{R}\mathsf{A}$ coverage on $\mathcal{P}\mathsf{rv}$ includes executable code residing in RAM or some non-volatile memory. $\mathcal{R}\mathsf{A}$ may also cover non-executable regions on $\mathcal{P}\mathsf{rv}$, i.e., data. Let M, of bit-size L, represent $\mathcal{P}\mathsf{rv}$'s memory to be attested. If the content of M is known *a priori* to $\mathcal{V}\mathsf{rf}$ and expected to be immutable, then $\mathcal{P}\mathsf{rv}$ can compute the authenticated integrity check over M and send the result to $\mathcal{V}\mathsf{rf}$, which can easily validate it. The same applies if M is mutable and its entropy is low: $\mathcal{V}\mathsf{rf}$ can compute (or pre-compute) all possible valid (benign) attestation results over M and thus validate $\mathcal{P}\mathsf{rv}$'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 $\mathcal{P}\mathsf{rv}$ to return to $\mathcal{V}\mathsf{rf}$ 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), $\mathcal{P}\mathsf{rv}$ can return the fixed-size measurement result produced by the authenticated integrity check 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 $\mathcal{V}\mathsf{rf}$, $\mathcal{P}\mathsf{rv}$ can easily zero it out before computing the authenticated integrity check. This makes it impossible for malware to hide in such regions, and obviates the need for $\mathcal{P}\mathsf{rv}$ to send $\mathcal{V}\mathsf{rf}$ an explicit copy of D.

3.1.3 $\mathcal{R}\mathsf{A}$ Timing Overhead

Timing complexity (overhead) of $\mathcal{R}\mathsf{A}$ on $\mathcal{P}\mathsf{rv}$ is dominated by the authenticated integrity check (or measurement) of attested memory, which, in turn, depends on the size of that memory, $\mathcal{P}\mathsf{rv}$'s computational capabilities, and underlying cryptographic function(s). One natural way to obtain a measurement is by computing a Message Authentication Code (MAC), based either on hashing (e.g., HMAC-SHA-2 [50]) or encryption (e.g., AES-CBC-MAC [41]). 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 [55] or EC-DSA [42]. MACs are 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 signed. Of course, for small memory sizes, signature computation is the dominating step. 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 the actual data.

We now illustrate some concrete timing measurements. As a sample $\mathcal{P}\mathsf{rv}$ hardware platform, we use ODROID-XU4 [37], a popular single-board MCU representative of medium-to-low-end embedded systems. Figure 3.2 shows timings of the measurement process 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,

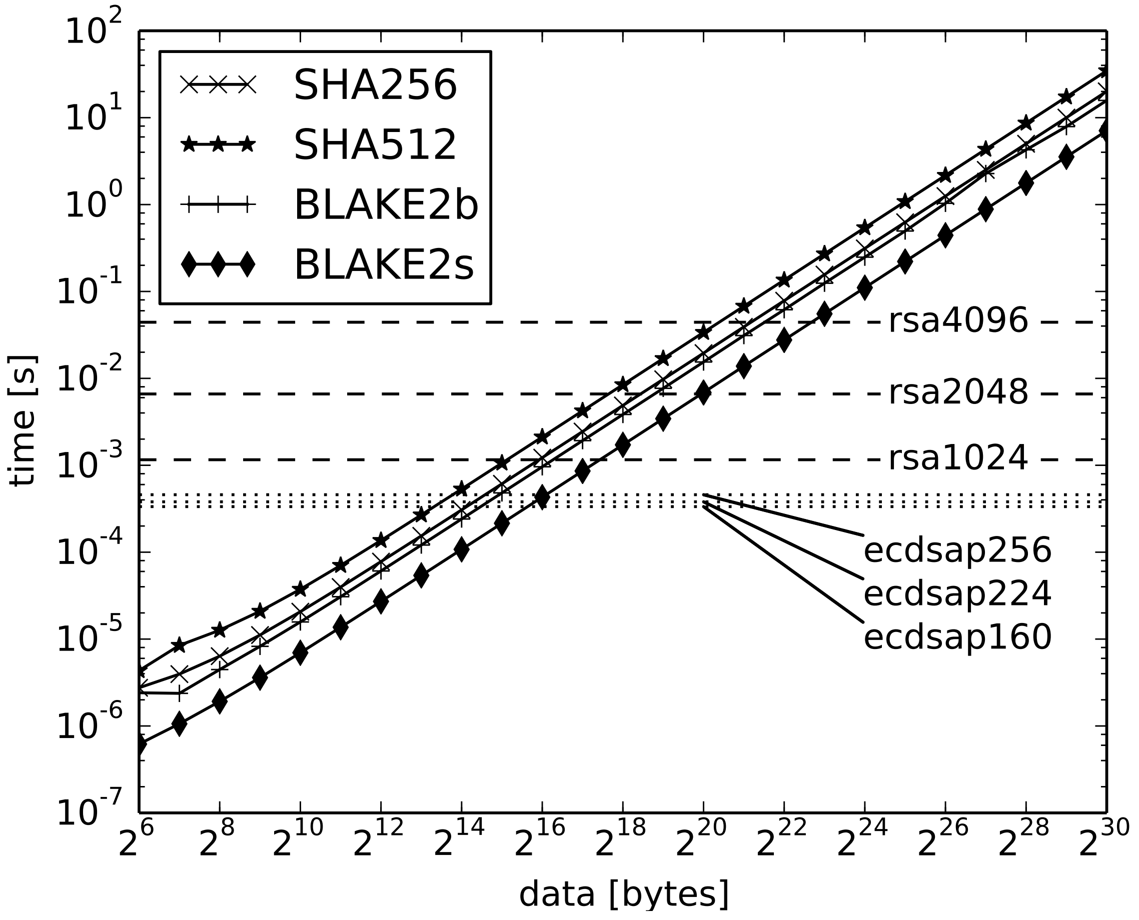


Figure 3.2: Timings of several hash functions and signatures on ODROID-XU4.

RSA-4096, ECDSA-160, ECDSA-224, and ECDSA-256.

As illustrated in Figure 3.2, for input sizes over 1MB, the measurement process takes longer than 0.01sec, and the cost of most signature algorithms become relatively 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 time-consuming and requires nearly 14sec.

3.2 $\mathcal{R}$A in Safety-Critical Settings

Since low/medium-end devices are often used in real-time and safety-critical applications, it is important to minimize the impact of $\mathcal{R}$A on normal operations (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 3.1.3.

For example, consider a sensing-actuating 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 read by its temperature sensor and triggers an alarm whenever that value exceeds a certain threshold. Given this safety-critical function, software integrity of $\mathcal{P}$rv is periodically validated via $\mathcal{R}$A, where the role of $\mathcal{V}$rf is played by a fire-alarm controller or a building's smart control panel. Upon receipt of a request from $\mathcal{V}$rf, the measurement process interrupts the safety-critical application and takes over. Using a hybrid technique (e.g., SMART), the measurement process must run uninterrupted in order to accurately reflect $\mathcal{P}$rv's current state. Assuming attested memory size of 1GB, the measurement process would run for approximately 7sec. However, if an actual fire breaks out soon after the measurement process starts, it would take a relatively long time for the application to regain control, sense the fire, and then trigger the alarm. Precious time lost as a result of the non-interruptible measurement process might have disastrous consequences.

At this point, it may be natural to conclude that the atomicity requirement should be relaxed and that the measurement process should be interruptible by a legitimate time-critical application. Unfortunately, allowing interrupts to the execution of the attestation code opens the door for malware to evade detection. For example, if $\mathcal{P}$rv is compromised, its safety-critical application may contain malware which presumably attempts to avoid detection. When confronted with imminent attestation and thus subsequent detection, it

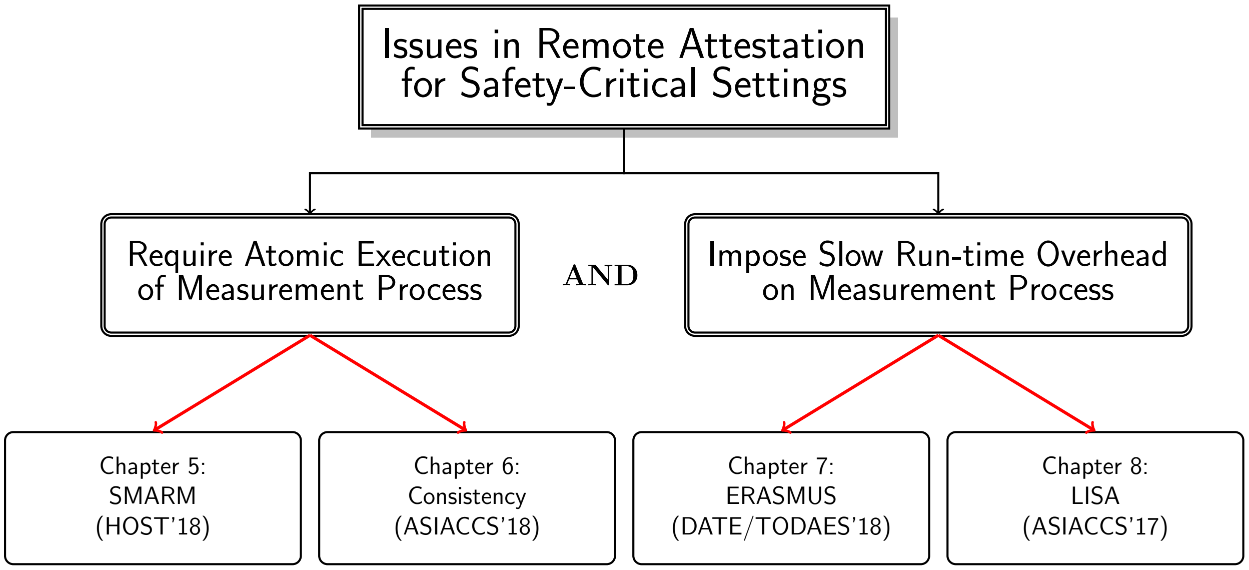

Issues in Remote Attestation for Safety-Critical Settings
Require Atomic Execution of Measurement Process
AND
Impose Slow Run-time Overhead on Measurement Process
Chapter 5: SMARM (HOST'18)
Chapter 6: Consistency (ASIACCS'18)
Chapter 7: ERASMUS (DATE/TODAES'18)
Chapter 8: LISA (ASIACCS'17)

Figure 3.3: Overview of potential solutions. Red arrow represents a mitigation method for a specific sub-problem.

may:

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

Therefore, we argue that there is an inherent conflict between the needs of safety-critical applications and $\mathcal{R}\mathsf{A}$ security requirements. Resolving this conflict represents a major challenge that we explore in the remainder of this dissertation.

3.3 Overview of Proposed Solutions

Our mitigation methods are summarized in Figure 3.3. One possible direction to resolve this conflict is to carefully relax the atomicity requirement while retaining all $\mathcal{R}\mathsf{A}$ security properties. This idea is pursued in Chapters 5 and 6 of this dissertation. Specifically, Chapter 5 allows $\mathcal{R}\mathsf{A}$ to be interruptible but enforces memory to be measured in a random (and secret) order. Chapter 6 presents an alternative approach that permits interrupts while ensuring the validity of secure $\mathcal{R}\mathsf{A}$ by locking memory, i.e., making it unmodifiable, during $\mathcal{R}\mathsf{A}$. It also presents various locking mechanisms, each providing different degrees of availability of writable memory.

The other issue investigated in this dissertation is how to resolve a conflict stemmed from a real-time computation overhead of the attestation process on $\mathcal{P}\mathsf{rv}$. This idea is treated in Chapters 7 and 8. Chapter 7 introduces the concept of periodic self-measurements, which results in more flexible scheduling of the measurement process. Using this approach, $\mathcal{P}\mathsf{rv}$ can avoid the conflict by safely and securely aborting the measurement process whenever safety-critical applications need to execute. This is possible because this approach imposes no real-time computation on $\mathcal{P}\mathsf{rv}$ (as part of its interaction with $\mathcal{V}\mathsf{rf}$). Chapter 8 presents two cRA schemes, and discuss how to integrate periodic self-measurement techniques into these two schemes.

In the next chapter, we present a hybrid $\mathcal{R}\mathsf{A}$ architecture, called HYDRA, that will serve as a building block when designing and implementing these mitigation techniques.

Chapter 4

Remote Attestation Using Formally Verified Microkernel

4.1 Introduction

This chapter introduces *the first* hybrid $\mathcal{R}\mathsf{A}$ design – called HYDRA – based upon formally verified components to provide memory isolation and protection guarantees. Our main rationale is that designing $\mathcal{R}\mathsf{A}$ techniques based upon such components increases confidence in security of such designs and their implementations. Of course, ideally, one would formally prove security of the entirety of a $\mathcal{R}\mathsf{A}$ system, as opposed to proving security separately for each component and then proving that its composition is secure. However, we believe that this is not yet possible given the current state of developments and capabilities in (automated) formal verification and synthesis of hardware and software.

One recent prominent example illustrating difficulty of correctly designing and implementing security primitives (especially, those blending software and hardware) is the TrustZone-based Qualcomm Secure Execution Environment (QSEE) kernel vulnerability and exploit reported in CVE-2015-6639 [62]. ARM TrustZone [6] is a popular System-on-Chip (SoC) and a CPU system-wide approach to security; it is adopted in billions of processors on various platforms, including: smartphones, tablets, personal computers, wearables and enterprise systems. CVE-2015-6639 enables privilege escalation and allows execution of code in the TrustZone kernel which can then be used to achieve undesired outcomes and expose keying material. This vulnerability was used to break Android's Full Disk Encryption (FDE) scheme by recovering the master keys [51]. This example demonstrates the difficulty of getting both the design and the implementation right; it also motivates the use of formally verified building blocks, which can yield more secure $\mathcal{R}\mathsf{A}$ techniques. To this end, our $\mathcal{R}\mathsf{A}$ design uses the formally verified seL4 microkernel to obtain memory isolation and access control. Such features have been previously attained with hardware in designs such as [31] and [47]. Using seL4 requires fewer hardware modifications to the underlying microprocessor and provides an automated formal proof of isolation guarantees of the implementation of the microkernel. *To the best of our knowledge, this is the first attempt to design and implement* $\mathcal{R}\mathsf{A}$ *using a formally verified microkernel.*

Our main goal is to investigate a previously unexplored segment of the design space of hybrid $\mathcal{R}\mathsf{A}$ schemes, specifically, techniques that incorporate formally verified and proven (using automated methods) components, such as the seL4 microkernel. Beyond using seL4 in our design, our implementation is also based on the formally verified executable of seL4; that executable is guaranteed to adhere to the formally verified and proven design. Another important goal, motivation and feature of our design is the expanded scope of efficient hybrid $\mathcal{R}\mathsf{A}$ techniques. While applicability of prominent prior results (particularly, SMART [31] and TrustLite [47]) is limited to very simple single-process low-end devices, we target more capable devices that can run multiple processes and threads. We believe that our work represents an important and necessary step towards building efficient hybrid $\mathcal{R}\mathsf{A}$ techniques upon solid and verified foundations. Admittedly, we do not verify our entire design and prove its security using formal methods. However, we achieve the next best thing by taking advantage of already-verified components and carefully arguing security of the overall design, considering results on systematic analysis of features required for securely realizing hybrid $\mathcal{R}\mathsf{A}$ [33]. To achieve our goals we make two main contributions: (1) design of HYDRA – the first hybrid $\mathcal{R}\mathsf{A}$ technique based on the formally verified seL4 microkernel which provides memory isolation and access control guarantees, (2) implementations of HYDRA on two commercially available development boards (Sabre Lite and ODROID-XU4) and their analysis via experiments to demonstrate practicality of the proposed design. We show that HYDRA can attest 10MB of memory in less than 250ms when using Speck [73] as the underlying block-cipher to compute a cryptographic checksum (MAC).

This section overviews HYDRA and its design rationale, discusses security objectives and features, as well as the adversarial model. Our notation used in this chapter is summarized below.

$\mathcal{A}dv$	Adversary
$\mathcal{P}\mathsf{rv}$	Prover
$\mathcal{V}\mathsf{rf}$	Verifier
PR_{Att}	Attestation Process on $\mathcal{P}\mathsf{rv}$
BC_{Att}	Attestation Code/Binary on $\mathcal{P}\mathsf{rv}$
K	Symmetric secret key shared by $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$

Table 4.1: Notation

4.1.1 Design Rationale

Our main objective is to explore a new segment of the overall $\mathcal{R}\mathsf{A}$ design space. The proposed hybrid $\mathcal{R}\mathsf{A}$ design – HYDRA – requires very little in terms of secure hardware and builds upon the formally verified seL4 microkernel. As shown in Section 4.3, the only hardware support needed by HYDRA is a hardware-enforced secure boot feature, which is readily available on commercial off-the-shelf development boards and processors, e.g., Sabre Lite boards. The rationale behind our design is that seL4 offers certain guarantees (mainly process isolation and access control to memory and resources) that provide $\mathcal{R}\mathsf{A}$ features that were previously feasible only using hardware components. In particular, what was earlier attained using additional MCU controls and Read-Only Memory (ROM) in the SMART [31] and TrustLite [47] architectures can now be instantiated using capability controls in seL4.

To motivate and justify the design of HYDRA, we start with the result of Francillon, et al. [33]. It provides a systematic treatment of $\mathcal{R}\mathsf{A}$ by developing a semi-formal definition of $\mathcal{R}\mathsf{A}$ as a distinct security service, and systematically de-constructing it into a necessary and sufficient security objective, from which specific properties are derived. These properties are then mapped into a collection of hardware and software components that results in an overall secure $\mathcal{R}\mathsf{A}$ design. Below, we summarize the security objective in $\mathcal{R}\mathsf{A}$ and its derived security properties. Sections 4.2 and 4.3 show how the security objective and properties are satisfied in HYDRA and instantiated in two concrete prototypes based on Sabre Lite and ODROID-XU4 boards.

4.1.2 Hybrid $\mathcal{R}\mathsf{A}$ Objective and Properties

According to [33], the security objective of $\mathcal{R}\mathsf{A}$ is to allow a (remote) prover ($\mathcal{P}\mathsf{rv}$) to create an unforgeable authentication token, that convinces a verifier ($\mathcal{V}\mathsf{rf}$) that the former is in some well-defined (expected) state. Whereas, if $\mathcal{P}\mathsf{rv}$ has been compromised (i.e., malware is present), the authentication token must reflect this. The work in [33] describes a combination of platform features that achieve aforementioned security objective and derives a set of properties both necessary and sufficient for secure $\mathcal{R}\mathsf{A}$. The conclusion of [33] is that the following properties collectively represent the minimal requirements to achieve secure $\mathcal{R}\mathsf{A}$ on any platform.

- *Exclusive Access to Attestation Key (K):* the attestation process (PR_{Att}) must have exclusive access to K. This is the most difficult requirement for (especially, low-end and mid-range) embedded devices. As argued in [33], this property is unachievable without some hardware support on low-end devices. If the underlying processor supports multiple privilege modes and a full-blown memory separation for each process, one could use a privileged process to handle any computation that involves K. However, low-end and mid-range processors generally do not offer such "luxury" features.

- *No Leaks:* no information related to (or derived from) K must be accessible after execution of PR_{Att}. To achieve this, all intermediate values that depend on K – except the final attestation token to be returned to $\mathcal{V}\mathsf{rf}$ – must be securely erased. This is applicable to very low-end devices, with none (or minimal) OS support and assuming that memory is shared between processes. However, if the underlying hardware and/or software guarantees strict memory separation among processes, this property is trivially satisfied.

- *Immutability:* To ensure that the attestation executable (BC_{Att}) cannot be modified, previous efforts (e.g., [31] or [33]) place it in and execute it from ROM, which is available on most, even low-end, platforms. ROM is a relatively inexpensive way to enforce BC_{Att}'s code immutability. Whereas, if the OS guarantees: (1) run-time process memory separation, and (2) immutability of BC_{Att} code (e.g., by checking its integrity/authenticity prior to execution), then BC_{Att} can reside, and be executed, in RAM.

- *Uninterruptability:* Execution of BC_{Att} must be uninterruptible. This is necessary to ensure that malware does not obtain the key (or some function thereof) by interrupting BC_{Att} while any key-related values remain in registers or other locations. SMART achieves this property via MCU controls. However, if PR_{Att} runs with the highest possible priority, the OS can ensure uninterruptibility.

- *Controlled Invocation (aka Atomicity):* BC_{Att} must only be invocable from its first instruction and must exit only at one of its legitimate last (exit) instruction. This is motivated by the need to prevent code-reuse attacks. As before, enforcing this property via MCU access controls can be replaced by OS support.

The work in [33] stipulates one extra property: *Secure Reset*, initiated whenever an attempt is detected to execute BC_{Att} from the middle of its code. We argue that this is not needed if controlled invocation is enforced. It suffices to raise an exception, as long as the memory space of PR_{Att} is protected and integrity of executable is guaranteed.

Another important security feature identified in [12] is to protect $\mathcal{P}\mathsf{rv}$ from $\mathcal{V}\mathsf{rf}$ impersonation as well as denial-of-service (DoS) attacks that attempt to forge, replay, reorder or delay attestation requests. All such attacks aim to maliciously invoke $\mathcal{R}\mathsf{A}$ functionality on $\mathcal{P}\mathsf{rv}$ and thus deplete $\mathcal{P}\mathsf{rv}$'s resources or take them away from its main tasks. According to [12], the following additional property is required:

- $\mathcal{V}\mathsf{rf}$ *Authentication:* PR_{Att} on $\mathcal{P}\mathsf{rv}$ must: (1) authenticate $\mathcal{V}\mathsf{rf}$ and (2) detect replayed, re-ordered and delayed requests. To achieve (1), the very same K can used to generate (by $\mathcal{V}\mathsf{rf}$) and verify (by $\mathcal{P}\mathsf{rv}$) all attestation requests. To satisfy (2), the work in [12] requires an additional hardware component: a reliable real-time clock. This clock must be loosely synchronized with $\mathcal{V}\mathsf{rf}$'s clock and must be write-protected.

4.1.3 Adversarial Model & Other Assumptions

Based on the recent taxonomy in [2], adversary ($\mathcal{A}dv$) in the context of $\mathcal{R}\mathsf{A}$ can be categorized as follows:

- *Remote:* exploits vulnerabilities in $\mathcal{P}\mathsf{rv}$'s software to inject malware, over the network.
- *Local:* located sufficiently near $\mathcal{P}\mathsf{rv}$ in order to eavesdrop on, and manipulate, $\mathcal{P}\mathsf{rv}$'s communication channel(s).

- *Physical:* has full (local) physical access to $\mathcal{P}\mathsf{rv}$ and its hardware; can perform physical attacks, e.g., use side channels to derive keys, physically extract memory values, and modify various hardware components.

[33] and [12] conclude that any $\mathcal{R}\mathsf{A}$ that satisfies all properties described in 4.1.2 always yields correct attestation tokens (i.e., no false positives and no false negatives) while achieving resilience to DoS attacks even in the presence of remote and local $\mathcal{A}dv$-s. HYDRA builds on top of these properties and similarly considers remote and local $\mathcal{A}dv$-s; physical $\mathcal{A}dv$ is considered to be out-of-scope.

We note that, at least in a single-prover setting[1], protection against physical attacks can be attained by encasing the CPU in tamper-resistant coating and employing standard techniques to prevent side-channel key leakage. These include: anomaly detection, internal power regulators and additional metal layers for tamper detection. We consider $\mathcal{P}\mathsf{rv}$ to be a (possibly) unattended remote hardware platform running multiple processes on top of seL4. Once $\mathcal{P}\mathsf{rv}$ boots up and runs in steady state, $\mathcal{A}dv$ might be in complete control of all application software (including code and data) before and after execution of PR_{Att}. Since physical attacks are out of scope, $\mathcal{A}dv$ can not induce hardware faults or retrieve K using side channels. Finally, recall that $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$ must share at least one secret key K. This key can be preloaded onto $\mathcal{P}\mathsf{rv}$ at installation time and stored as part of BC_{Att}. We do not address the details of this procedure.

4.2 Design

This section overviews seL4 and discusses its use in HYDRA. It then describes the sequence of operations in HYDRA.

[1] See [38] for physical attack resilience in groups of provers.

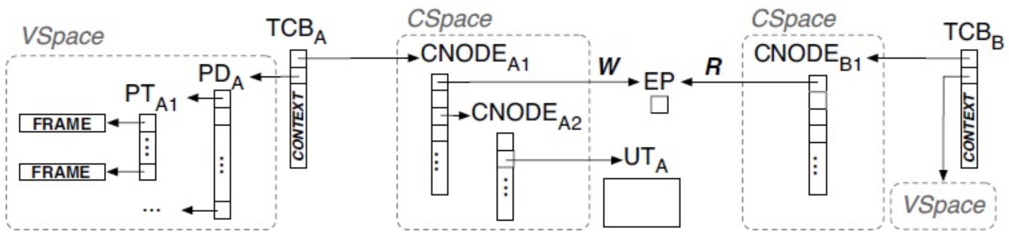


Figure 4.1: Sample seL4 instantiation from [81].

4.2.1 seL4 Overview

seL4 is a member of the L4 microkernel family, specifically designed for high-assurance applications by providing isolation and memory protection between different processes. These properties are mathematically guaranteed by a full-code level functional correctness proof, using automated tools. A further correctness proof of the C code translation is presented in [82], thus extending functional correctness properties to the binary level without needing a trusted compiler. Therefore, behavior of the seL4 binary strictly adheres to, and is fully captured by, the abstract specifications.

Similar to other operating systems, seL4 divides the virtual memory into two separated address spaces: *kernel-space* and *user-space*. The kernel-space is reserved for the execution of the seL4 microkernel while the application software is run in user-space. By design, and adhering to the nature of microkernels, the seL4 microkernel provides minimal functionalities to user-space applications: thread, inter-process communication (IPC), virtual memory, capability-based access control and interrupt control. The seL4 microkernel leaves the implementations of other traditional operating system functions – such as device drivers and file systems – to user-space.

Figure 4.1 (borrowed from [81]) shows an example of seL4 instantiation with two threads – sender A and receiver B – that communicate via an *EndPoint* EP. Each thread has a *Thread Control Block* (TCB) that stores its context, including: stack pointer, program counter, register values, as well as pointers to *Virtual-address Space* (VSpace) and *Capability Space* (CSpace). VSpace represents available memory regions that the seL4 microkernel allocated to each thread. The root of VSpace represents a *Page Directory* (PD), which contains *Page Table* (PT) objects. *Frame* object representing a region of physical memory resides in a PT. Each thread also has its own kernel managed CSpace used to store a *Capability Node* (CNode) and *capabilities*. CNode is a table of slots, where each slot represents either a capability or another CNode.

A capability is an unforgeable token representing an access control authorization of each kernel object or component. A thread cannot directly access or modify a capability since CSpace is managed by, and stored inside, the kernel. Instead, a thread can invoke an operation on a kernel object by providing a pointer to a capability that has sufficient authority for that object to the kernel. For example, sender A in Figure 4.1 needs a write capability of EP for sending a message, while receiver B needs a read capability to receive a message. Besides read and write, *grant* is another access right in seL4, available only for an endpoint object. Given possession of a grant capability for an endpoint, any capability from the possessor can be transferred across that endpoint. For instance, if A in Figure 4.1 has grant access to EP, it can issue one of its capabilities, say a frame, to B via EP. Also, capabilities can be statically issued during a thread's initialization by the *initial process*. The initial process is the first executable user-space process loaded into working memory (i.e., RAM) after the seL4 microkernel is loaded. This special process then forks all other processes. Section 4.2.4 describes the role, the details and the capabilities of the initial process in HYDRA design.

seL4's main "claim to fame" is in being the first formally verified general-purpose operating system. Formal verification of the seL4 microkernel is performed by interactive, machine-assisted and machine-checked proof using a theorem prover Isabelle/HOL. Overall functional correctness is obtained through a *refinement* proof technique, which demonstrates that the binary of seL4 refines an abstract specification through three layers of refinement. Consequently, the seL4 binary is fully captured by the abstract specifications. In particular, two important feature derived from seL4's abstract specifications, are that: **the kernel never crashes**. Another one is that: **every kernel API call always terminates and returns to user-space**. Comprehensive details of seL4's formal verification can be found in [44].

Another seL4 feature very relevant to our work is: **correctness of access control enforcement** derived from functional correctness proof of seL4. [81] and [56] introduce formal definitions of the access control model and information flow in seL4 at the abstract specifications. They demonstrate the refinement proof from these modified abstract specifications to the C implementation using Isabelle/HOL theorem prover, which is later linked to the binary level (by the same theorem prover). As a result, three properties are guaranteed by the access control enforcement proof: (1) *Authority Confinement*, (2) *Integrity* and (3) *Confidentiality*. Authority confinement means that authority propagates correctly with respect to its capability. For example, a thread with a read-only capability for an object can only read, and not write to, that object. Integrity implies that system state cannot be modified without explicit authorization. For instance, a read capability should not modify internal system state, while write capability should only modify an object associated with that capability. Finally, confidentiality means that an object cannot be read or inferred without a read capability. Thus, the proof indicates that access control in seL4, once specified at the binary level, is correctly enforced as long as the seL4 kernel is active.

We now show how seL4's access control enforcement property satisfies required $\mathcal{R}\mathsf{A}$ features.

4.2.2 Deriving seL4 Access Controls

We now describe access control configuration of seL4 user-space that achieves most required properties for secure $\mathcal{R}\mathsf{A}$, as described in section 4.1.2. We examine each feature and identify the corresponding access control configuration. Unlike prior hybrid designs, HYDRA pushes

Table 4.2: Security properties in hybrid $\mathcal{R}$A

Security Property	SMART [31]	TrustLite [47]	HYDRA
Exclusive Access to K	HW (Mod. Data Bus)	SW (programmed MPU)	SW (seL4)
No Leaks	SW (Instrumented SW)	HW (CPU Exception Engine)	SW (seL4)
Immutability	HW (ROM)	HW (ROM) and SW (MPU)	HW (ROM) and SW (seL4)
Uninterruptability	SW (Interrupt Disabled)	HW (CPU Exception Engine)	SW (seL4)
Controlled Invocation	HW (MCU Control)	SW (MPU)	SW (seL4)

almost all of these required features into software, as long as the seL4 microkernel boots correctly. (A comparison with SMART and TrustLite is in Table 4.2.)

- *Exclusive Access to K* is directly translated to an access control configuration. Similar to previous hybrid approaches, K can be hard-coded into the BC_{Att} at production time. Thus, BC_{Att} needs to be configured to be accessible only to PR_{Att}.

- *No Leaks* is achieved by the separation of virtual address space. Specifically, the virtual memory used for K-related computation needs to be configured to be accessible to only PR_{Att}.

- *Immutability* is achieved using combination of verifiable boot and runtime isolation guarantee from seL4. At runtime, BC_{Att} must be immutable, which can be guaranteed by restricting the access control to the executable to only PR_{Att}. However, this is not enough to assure immutability of BC_{Att} executable because BC_{Att} can be modified after loaded into RAM but before executed. Hence, a verifiable boot of BC_{Att} is required.

- *Uninterruptability* is ensured by setting the scheduling priority of PR_{Att} higher than other processes since the formal proof of seL4 scheduling mechanism guarantees that a lower priority process cannot preempt the execution of a higher priority process. In addition, seL4 guarantees that, once set, the scheduling priority of any process can not be increased at runtime.

Note that this feature implies that PR_{Att} needs to be the initial user-space process since the seL4 microkernel always assigns the highest priority to the initial process.

- *Controlled Invocation* is achieved by the isolation of process' execution. In particular, TCB of PR_{Att} cannot be accessed or manipulated by other processes.
- $\mathcal{V}\mathsf{rf}$ *Authentication* is achieved by configuring a capability of the real-time clock to be read-only for other user-space processes.

With these features, we conclude that the access control configuration of seL4 user-space needs to (at least) include the following:

$\boxed{\textbf{C1}}$ PR_{Att} has exclusive access to BC_{Att}; this also includes K residing in BC_{Att}. (Recall that PR_{Att} is the attestation **process**, while BC_{Att} is the executable that actually performs attestation.)

$\boxed{\textbf{C2}}$ PR_{Att} has exclusive access to its TCB.

$\boxed{\textbf{C3}}$ PR_{Att} has exclusive access to its VSpace.

$\boxed{\textbf{C4}}$ PR_{Att} has exclusive write-access to the real-time clock.

Even though this access control configuration can be enforced at the binary code level, this assumption is based on that seL4 is loaded into RAM correctly. However, this can be exploited by an adversary by tricking the boot-loader to boot his malicious seL4 microkernel instead of the formally verified version and insert a new configuration violating above access controls. Thus, the hardware signature check of the seL4 microkernel code is required at boot time. The similar argument can also be made for BC_{Att} code. As a result, additional integrity check of BC_{Att} code needs to be performed by seL4 before executing.

4.2.3 Building Blocks

In order to achieve all security properties described above, HYDRA requires the following four components.

- **Read-Only Memory:** region primarily storing immutable data (e.g. hash of public keys or signature of software) required for secure boot of the seL4 microkernel.
- **MCU Access Control Emulation:** high-assurance software framework capable of emulating MCU access controls to attestation key K. At present, seL4 is the only formally verified and mathematically proven microkernel capable of this task.
- **Attestation Algorithm:** software residing in PR_{Att} and serving two main purposes: authenticating an attestation request, and performing attestation on memory regions.
- **Reliable Real-Time Clock:** loosely synchronized (with $\mathcal{V}\mathsf{rf}$) real-time clock. This component is required for mitigating denial-of-service attacks that involve $\mathcal{V}\mathsf{rf}$ impersonation (via replay, reorder and delay)[12]. If $\mathcal{P}\mathsf{rv}$ does not have a clock, a secure counter can replace a real-time clock with the downside of delayed message detection. We also note that this component needs not be write-protected at hardware level due to a capability-based access control guarantee from seL4.

4.2.4 Sequence of Operation

The sequence of operations in HYDRA, shown in Figure 4.2, has three steps: boot, setup, and attestation.

Boot Process. Upon a boot, $\mathcal{P}\mathsf{rv}$ first executes a ROM-resident boot-loader. The boot-loader verifies authenticity and integrity of the seL4 microkernel binary. Assuming this

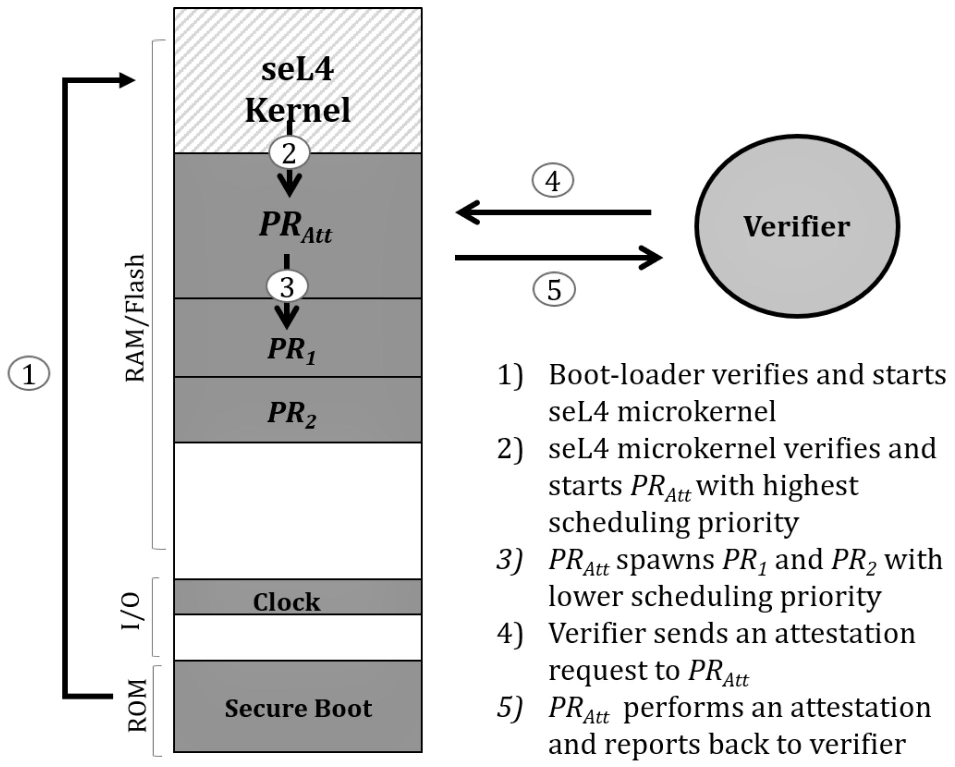

seL4 Kernel
PR_{Att}
PR_1
PR_2
Clock
Secure Boot
RAM/Flash
I/O
ROM
Verifier
1) Boot-loader verifies and starts seL4 microkernel
2) seL4 microkernel verifies and starts PR_{Att} with highest scheduling priority
3) PR_{Att} spawns PR_1 and PR_2 with lower scheduling priority
4) Verifier sends an attestation request to PR_{Att}
5) PR_{Att} performs an attestation and reports back to verifier

Figure 4.2: Sequence of operation in HYDRA

verification succeeds, the boot-loader loads all executables, including kernel and user-space, into RAM and hands over control to the seL4 microkernel. Further details of secure boot in our prototype can be found in Section 4.3.

seL4 Setup. The first task in this step is to have the seL4 microkernel setting up the user-space and then starting PR_{Att} as the initial user-space process. Once the initialization inside the kernel is over, the seL4 microkernel gathers capabilities for all available memory-mapped locations and assigns them to PR_{Att}. The seL4 kernel also performs an authenticity and integrity check of PR_{Att} to make sure that it has not been modified. After successful authentication, the seL4 microkernel passes control to PR_{Att}.

With full control over the system, PR_{Att} starts the rest of user-space with a lower scheduling priority and distributes capabilities that do not violate the configuration specified earlier. After completing configuration of memory capabilities and starting the rest of the user-space,

PR_{Att} initializes the network interface and waits for an attestation request.

Attestation. An attestation request, sent by a verifier, consists of 4 parameters: (1) T_R reflecting $\mathcal{P}\mathsf{rv}$'s time when the request was generated, (2) target process p, (3) its memory range $[a, b]$ that needs to be attested, and (4) cryptographic checksum C_R of the entire attestation request.

Similar to SMART [31], the cryptographic checksum function used in attestation is implemented as a Message Authentication Code (MAC), to ensure authenticity and integrity of attestation protocol messages.

Upon receiving an attestation request PR_{Att} checks whether T_R is within an acceptable range of the $\mathcal{P}\mathsf{rv}$'s real-time clock before performing any cryptographic operation; this is in order to mitigate potential DoS attacks. If T_R is not fresh, PR_{Att} ignores the request and returns to the waiting state. Otherwise, it verifies C_R. If this fails, PR_{Att} also abandons the request and returns to the waiting state.

Once the attestation request is authenticated, PR_{Att} computes a cryptographic checksum of the memory region $[a,\ b]$ of process p. Finally, PR_{Att} returns the output to $\mathcal{V}\mathsf{rf}$. The pseudo-code of this process is shown in Algorithm 1.

4.3 Implementation

To demonstrate feasibility and practicality of HYDRA, we developed two prototypes on commercially available hardware platforms: ODROID-XU4 [37] and Sabre Lite [10]. We focus on the latter, because of lack of seL4 compatible network drivers and programmable ROM in current ODROID-XU4 boards. We provide details of, and insight gained from, our implementation, mostly concerning secure initialization and the configuration of the access

Algorithm 1: BC_{Att} pseudo-code

```
Input  : $T_R$ timestamp of request
         $p$ target process for attestation
         $a, b$ start/end memory region of target process
         $C_R$ cryptographic checksum of request
Output: Attestation Report
begin
    /* Check freshness of timestamp and verify request */
    if $\neg$ CheckFreshness($T_R$) then
        exit();
    end
    if $\neg$ VerifyRequest($C_R$, $K$, $T_R \| p \| a \| b$) then
        exit();
    end
    /* Retrieve address space of process $p$ */
    $Mem \leftarrow$ RetrieveMemory($p$);
    /* Compute attestation report */
    MacInit($K$);
    MacUpdate($T_R \| p \| a \| b$);
    for $i \in [a, b]$ do
        MacUpdate($Mem[i]$);
    end
    $out \leftarrow$ MacFinal();
    return $out$
end
```

control in HYDRA; we then describe the implementation of key storage and timestamp generation. Section 4.5 presents a detailed performance evaluation of the implementation.

4.3.1 seL4 User-space Implementation

Our prototype is implemented on top of version 1.3 of the seL4 microkernel [59]. The complete implementation, including helper libraries and the networking stack, consists of $105,360$ lines of C code (see Table 4.3 for a more detailed breakdown). The overall size of executable is 574KB whereas the base seL4 microkernel size is 215KB. Excluding all helper libraries, the implementation of HYDRA is just 2800 lines of C code. In the user-space, we base our C code on following libraries: seL4utils, seL4vka and seL4vspace; these libraries

Table 4.3: Complexity of HYDRA impl. on our prototype

Complexity	HYDRA with net. and libs	HYDRA w/o net. stack	HYDRA w/o net. and libs	seL4 Kernel Only
LoC	105,360	68,490	11,938	9,142
Exec Size	574KB	476KB	N/A	215KB

provide the abstraction of processes, memory management and virtual space respectively. In our prototypes, PR_{Att} is the initial process in the user-space and receives capabilities to all memory locations not used by seL4. Other processes in user-space are spawned by this PR_{Att}. We also ensure that access control of those processes does not conflict with what we specified in Section 4.2. The details of this access control implementation are described below in this section.

The basic C function calls are implemented in muslc library. seL4bench library is used to evaluate timing and performance of our HYDRA implementation. For a timer driver, we rely on its implementation in seL4platsupport. All source code for these helper libraries can be found in [58] and these libraries contribute around 50% of the code base in our implementation. We use an open-source implementation of a network stack and an Ethernet driver in the user-space [57]. We argue that this component, even though not formally verified, should not affect security objective of HYDRA as long as an IO-MMU is used to restrict Direct Memory Access (DMA) of an Ethernet driver. The worst case that can happen from not formally verified network stack is symmetrical denial-of-service, which is out of scope of HYDRA.

4.3.2 Secure Boot Implementation

Here, we describe how we integrate an existing secure boot feature (in Sabre Lite) with our HYDRA implementation.

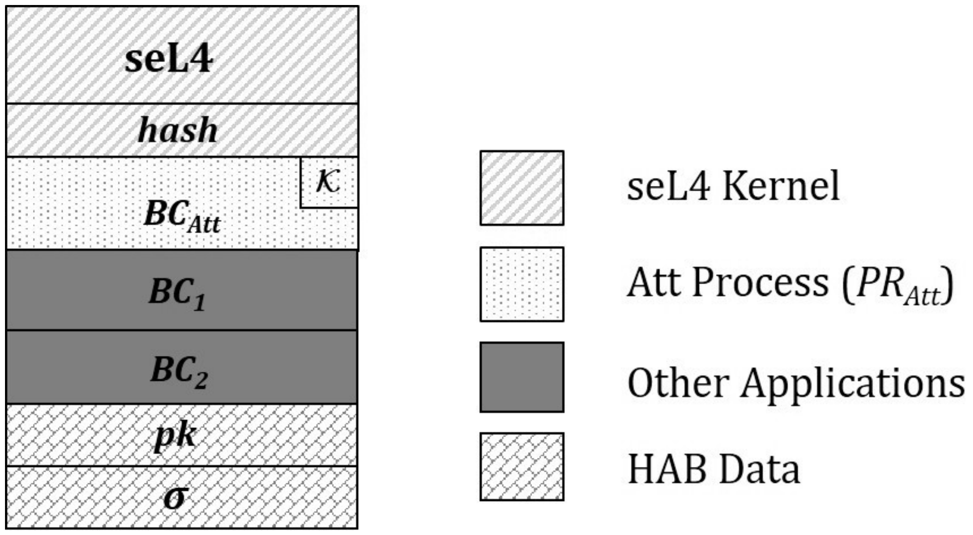


Figure 4.3: Image layout in flash

Secure Boot in Sabre Lite

NXP provides a secure boot feature for Sabre Lite boards, called High Assurance Boot (HAB) [34]. HAB is implemented based on a digital signature scheme with public and private keys. A private key is needed to generate a signature of the software image during manufacturing whereas a public key is used by ROM APIs for decrypting and verifying the software signature at boot time. A public key and a signature are attached to the software image, which is pre-installed in a flash during manufacturing. The digest of a public key is fused into a one-time programmable ROM in order to ensure authenticity of the public key and the booting software image. At boot time, the ROM boot-loader first loads the software image into RAM and then verifies the attached public key by comparing it with the reference hash value in ROM. It then authenticates the software image through the attached signature and the verified public key. Execution of this image is allowed only if signature verification succeeds. Without a private key, an adversary cannot forge a legitimate digital signature and thus is unable to insert and boot his malicious image.

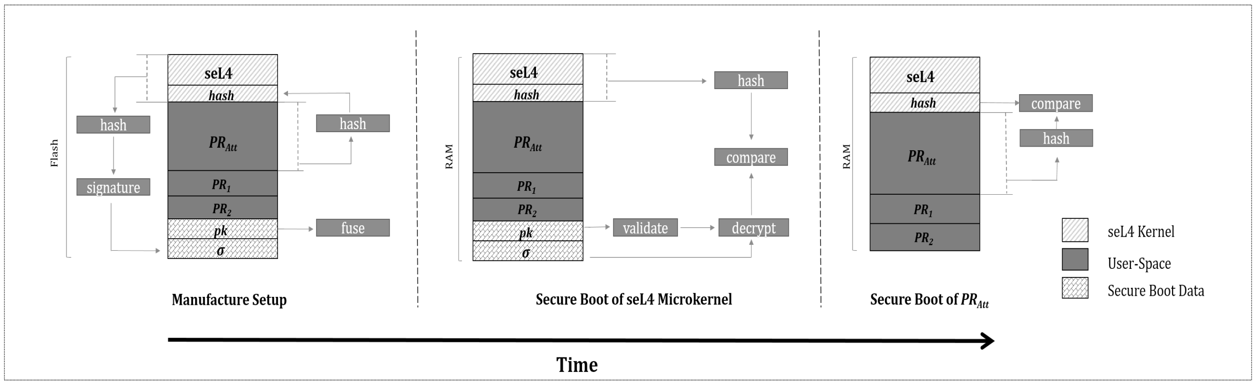


Figure 4.4: Secure boot sequence in SabreLite prototype

Secure Boot of HYDRA

HAB ensures that the seL4 microkernel is the first program initialized after the ROM boot-loader. This way, the entire seL4 microkernel binary code can be covered when computing the digital signature during manufacturing. Moreover, seL4 needs to be assured that it gives control of the user-space to the verified PR_{Att}, which means that seL4 has to perform an integrity check of PR_{Att} before launching it. Consequently, a hash of BC_{Att} needs to be included in the seL4 microkernel's binaries during production time and be validated upon starting the initial process.

With this procedure, a chain of trust is established in the $\mathcal{RA}$ system in HYDRA. This implies that no other programs, except the seL4 microkernel can be started by the ROM boot-loader and consequently only PR_{Att} is the certified initial process in the user-space, which achieve the goal of secure boot of $\mathcal{RA}$ system. Figure 4.4 illustrates the secure boot of HYDRA in Sabre Lite prototype.

4.3.3 Access Control Implementation

Here we describe how the access control configuration specified in section 4.2 is implemented in our HYDRA prototype. Our goal is to show that in the implementation of HYDRA no other user-space processes, except PR_{Att}, can have any kind of access to: (1) the binary executable code (including K), (2) the virtual address space of PR_{Att}, and (3) the TCB of PR_{Att}. To provide those access restrictions in the user-space, we make sure that we do not assign capabilities associated to those memory regions to other user-space processes. Recall that PR_{Att} as the initial process contains all capabilities to every memory location not used by the seL4 microkernel. And there are two ways for PR_{Att} to issue capabilities: dynamically transfer via endpoint with grant access right or statically assign during bootstrapping a new process.

In our implementation, PR_{Att} does not create any endpoint with grant access, which disallows any capability of PR_{Att} to transfer to a new process after created. Thus, the only way that capabilities can be assigned to a new process is before that process is spawned. When creating a new process, PR_{Att} assigns only minimal amount of capabilities required to operate that process, e.g. in our prototype, only the CSpace root node and fault endpoint (used for receiving IPCs when this thread faults) capabilities are assigned to any newly created process. Limited to only those capabilities, any other process cannot access the binary executable code as well as existing virtual memory and TCB of PR_{Att}.

Moreover, during bootstrapping the new process, PR_{Att} creates a new PD object serving as the root of VSpace in the new process. This is to ensure that any new process' virtual address space is initially empty and does not overlap with the existing virtual memory of PR_{Att}. Without any further dynamic capability distribution, this guarantees that other processes cannot access any memory page being used by PR_{Att}. Sample code for configuring a new process in our prototype is provided in Listing 4.1 below.

```
int sel4utils_configure_process_custom(sel4utils_process_t *process, vka_t *vka, vspace_t *spawner_vspace,
     sel4utils_process_config_t config)
{
    int error;
    sel4utils_alloc_data_t * data = NULL;
    memset(process, 0, sizeof(sel4utils_process_t));
    seL4_CapData_t cspace_root_data = seL4_CapData_Guard_new(0, seL4_WordBits - config.
        one_level_cspace_size_bits);
    process->own_vspace = config.create_vspace;
    error = vka_alloc_vspace_root(vka, &process->pd);
    if (error) {
        goto error;
    }
    if (assign_asid_pool(config.asid_pool, process->pd.cptr) != seL4_NoError) {
        goto error;
    }
    process->own_cspace = config.create_cspace;
    if (create_cspace(vka, config.one_level_cspace_size_bits, process, cspace_root_data) != 0) {
        goto error;
    }
    if (create_fault_endpoint(vka, process) != 0) {
        goto error;
    }
    sel4utils_get_vspace(spawner_vspace, &process->vspace, &process->data, vka, process->pd.cptr,
        sel4utils_allocated_object, (void *) process);
    process->entry_point = sel4utils_elf_load(&process->vspace, spawner_vspace, vka, vka, config.image_name);
    if (process->entry_point == NULL) {
        goto error;
    }
    error = sel4utils_configure_thread(vka, spawner_vspace, &process->vspace, SEL4UTILS_ENDPOINT_SLOT, config.
        priority, process->cspace.cptr, cspace_root_data, &process->thread);
    if (error) {
        goto error;
    }
    return 0;
error:
    /* clean up */
    ...
    return -1;
}
int sel4utils_configure_thread_config(vka_t *vka, vspace_t *parent, vspace_t *alloc, sel4utils_thread_config_t
     config, sel4utils_thread_t *res)
{
    memset(res, 0, sizeof(sel4utils_thread_t));
    int error = vka_alloc_tcb(vka, &res->tcb);
    if (error == -1) {
        sel4utils_clean_up_thread(vka, alloc, res);
        return -1;
    }
    res->ipc_buffer_addr = (seL4_Word) vspace_new_ipc_buffer(alloc, &res->ipc_buffer);
    if (res->ipc_buffer_addr == 0) {
        return -1;
    }
    if (write_ipc_buffer_user_data(vka, parent, res->ipc_buffer, res->ipc_buffer_addr)) {
        return -1;
    }
    seL4_CapData_t null_cap_data = {{0}};
    error = seL4_TCB_Configure(res->tcb.cptr, config.fault_endpoint, config.priority, config.cspace, config.
         cspace_root_data, vspace_get_root(alloc), null_cap_data, res->ipc_buffer_addr, res->ipc_buffer);
    if (error != seL4_NoError) {
        sel4utils_clean_up_thread(vka, alloc, res);
        return -1;
    }
    res->stack_top = vspace_new_stack(alloc);
    if (res->stack_top == NULL) {
        sel4utils_clean_up_thread(vka, alloc, res);
        return -1;
    }
    return 0;
}
```

Listing 4.1: Sample code for initializing a new process

4.3.4 Key Storage

Traditionally, in previous hybrid designs, $\mathcal{P}\mathsf{rv}$ requires a special hardware-controlled memory location for securely storing $\mathcal{K}$ and protecting it from software attacks. However, in HYDRA, it is possible to store $\mathcal{K}$ in a normal memory location (e.g. flash) due to the formally verified access control and isolation properties of seL4. Moreover, since $\mathcal{K}$ is stored in a writable memory, its update can easily happen without any secure hardware involvement. Thus, in our prototypes, $\mathcal{K}$ is hard-coded at production time and stored in the same region as BC_{Att}.

4.3.5 Mitigating Denial-of-Service Attacks

Our HYDRA prototype uses the same K for two purposes: (1) $\mathcal{P}\mathsf{rv}$ computing the attestation token, and (2) authenticating $\mathcal{V}\mathsf{rf}$ attestation requests. (Recall that K can be accessed only by PR_{Att}.) Alternatively, PR_{Att} can derive two separate keys from K, one for each purpose, through a key derivation function (KDF).

The work in [12] also shows that authenticating attestation requests is insufficient to mitigate DoS attacks since $\mathcal{A}dv$ can eavesdrop on genuine attestation requests and then delay or replay them. [12] concludes that timestamps, obtained from a reliable real-time clock (synchronized with $\mathcal{V}\mathsf{rf}$'s clock), are required in order to handle replay, reorder and delay attacks.

There are currently no real-time clock drivers available for seL4. Instead, we generate a pseudo-timestamp by a timer, the driver for which is provided by seL4platsupport, and a timestamp of the first validated request, as follows:

When a device first boots and securely starts PR_{Att} as the initial process, PR_{Att} loads a timestamp, T_0, that was previously saved (in a separated location in flash) before the last reset. When the first attestation request arrives, PR_{Att} checks whether its timestamp, $T_1 > T_0$ and, if so, proceeds to $VerifyRequest$. (Else, the request is discarded). Once the request is validated, PR_{Att} keeps track of T_1 and starts a counter. At any later time, a timestamp can be constructed by combining the current counter value with T_1. Also, PR_{Att} periodically generates and saves this timestamp value on flash, to be used after the next reboot. The prototype also ensures that the timestamp is write-protected by not assigning write capabilities for a memory region (storing T_0 and a timer device driver) to any other processes.

4.4 Security Analysis

We now informally show that HYDRA satisfies the minimal set of requirements to realize secure $\mathcal{R}\mathsf{A}$ (described in Section 4.1.2). HYDRA's key features are:

(1) seL4 is the first executable loaded in a HYDRA-based system upon boot/initialization. Correctness of this step is guaranteed by a ROM integrity check at boot time, e.g., HAB in the Sabre Lite case.

(2) PR_{Att} [2] is the initial user-space process loaded into memory and executed by seL4. This is also supported via a software integrity check performed by seL4 before spawning the initial process.

(3) PR_{Att} starts with the highest scheduling priority and never decreases its own priority value. This can be guaranteed by checking that BC_{Att} does not contain any system calls to decrease its priority.

(4) Any subsequent process executed by seL4 is spawned by P_{Attest} and does not get the highest scheduling priority. This can be ensured by inspecting BC_{Att} to check that all invocations of other processes are with a lower priority value. Once a process is loaded with a certain priority, seL4 prevents it from increasing its priority value; this is formally verified and guaranteed by seL4 implementation.

(5) The software executable and K can only be mapped into the address space of PR_{Att}. This is guaranteed by ensuring that in the BC_{Att} no other process on initialization (performed in PR_{Att}) receives the capabilities to access said memory ranges.

(6) Virtual memory used by PR_{Att} cannot be used by any other process; this includes any memory used for any computation involving the key, or related to other values computed

[2] PR_{Att} is different from BC_{Att} per Figure 4.3. PR_{Att} is what is called "initial process" in Figure 4.3 and it contains BC_{Att} executable as a component.

using the key. This is formally verified and guaranteed in the seL4 implementation.

(7) Other processes cannot control or infer execution of PR_{Att} (protected by exclusive capability to TCB's PR_{Att}).

(8) Access control properties, i.e., authority confinement, integrity and confidentiality, in seL4's binary are mathematically guaranteed by its formal verification.

(9) Other processes cannot modify or reset the real-time clock. This can be guaranteed by verifying that BC_{Att} does not give away a write capability of the clock to other processes.

Given the above features, the security properties in Section 4.1.2 are satisfied because:

Exclusive Access to K: (5), (6) and (8) guarantee that only PR_{Att} can have access to K.

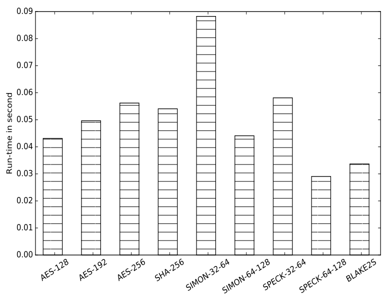
No Leaks: (6) and (8) ensure that intermediate values created by key-related computation inside PR_{Att} cannot be leaked to or learned by other processes.

Immutability: (1) and (2) imply that HYDRA is initialized into the correct expected known initial states and that the correct binary executable is securely loaded into RAM. (5) also prevents other processes from modifying that executable.

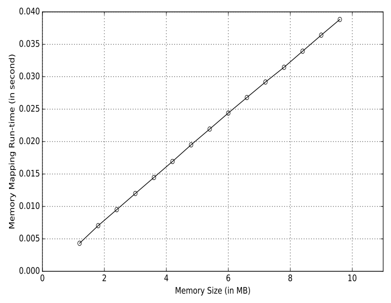
Uninterruptability: (3) and (4) guarantee that other processes, always having a lower priority value compared to PR_{Att}, cannot interrupt the execution of PR_{Att}.

Controlled Invocation: (7) ensures that the execution of PR_{Att} cannot be manipulated by other applications.

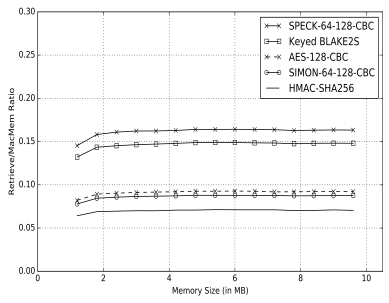
$\mathcal{V}$rf Authentication: (5), (6) and (8) ensure that K cannot be accessed and/or inferred by other processes. (8) and (9) ensure that no other process can modify and influence a timestamp value.



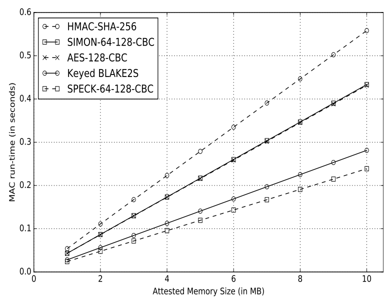
(a) MAC implementations



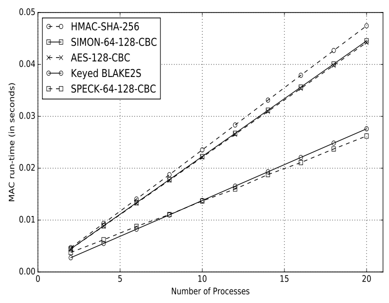
(b) Memory mapping in seL4



(c) MacMem vs retrieveMem



(d) MacMem vs mem size



(e) MacMem vs num processes

Figure 4.5: Evaluation of HYDRA in SabreLite prototype

4.5 Evaluation

Ideally, we would have liked to compare the performance of HYDRA with that of previous hybrid designs such as SMART and TrustLite on the same hardware platform. However, this is not feasible because SMART and TrustLite are designed for low-end micro-controllers and development platforms based on such micro-controllers (currently) cannot run seL4. In addition, SMART and TrusLite require some modifications to the micro-controller's hardware and are thus not available on off-the-shelf development platforms. We instead present performance evaluation of HYDRA using the commercially available I.MX6 Sabre Lite and ODROID-XU4 development platform.

4.5.1 Evaluation Results on I.MX6 Sabre Lite

We conduct experiments to assess speed of, and overhead involved in, performing attestation using different types of keyed Message Authentication Code (MAC) functions, on various numbers of user-space processes and sizes of memory regions to be attested. We obtain the fastest performance using the Speck MAC; HYDRA can attest 10MB in less than 250msec in that case. Our prototypes are implemented in C and compiled with an O2 flag.

Breakdown of Attestation Runtime. The attestation algorithm (Algorithm 1) is composed of three operations. $VerifyRequest$ (lines 3 to 9) is responsible for verifying an attestation request and whether it has been recently generated by an authorized verifier. $RetrieveMem$ (line 11) maps memory regions from a target process to PR_{Att}'s address space and returns a pointer to the mapped memory. $MacMem$ (lines 13 to 20) computes a cryptographic checksum (using K) on the memory regions.

As shown in Table 4.4, the runtime of $MacMem$ contributes the highest amount of the overall BC_{Att} runtime: 89% of total time for attesting 1MB of memory and 92% for attesting 20 KB of memory on Sabre Lite; whereas $RetrieveMem$ and $VerifyRequest$ together require less than 11% of the overall time.

Performance of RetrieveMem in seL4. Another important factor affecting the performance of HYDRA is the runtime of $RetrieveMem$: the time PR_{Att} takes to map the attested memory regions to its own virtual address space. As expected, Figure 4.5b illustrates the memory mapping runtime in seL4 is linear in terms of mapped memory size. In addition, we compare the runtime of $RetrieveMem$ and $MacMem$ on larger memory sizes. Figure 4.5c illustrates that the runtime ratio of $RetrieveMem$ to various implementations of $MacMem$ is always less than 20%. This confirms that retrieving memory and mapping it to the address space account for only a small fraction of the total attestation time in HYDRA. This illustrates that whatever overhead seL4 introduces when enforcing access control on memory

Table 4.4: Performance breakdown of Algorithm 1 on I.MX6-SabreLite @ 1GHz

Operations	1 MB of Memory		20 KB of Memory	
	Time in cycle	Proportion	Time in cycle	Proportion
$VerifyRequest$	1,604	<0.01%	1,604	0.29%
$RetrieveMem$	3,221,307	10.7%	45,624	8.21%
$MacMem$	26,880,057	89.29%	508,334	91.5%
$Overall$	30,102,968	100%	555,562	100%

is not significant and does not render HYDRA impractical.

Performance of MacMem in seL4. Since $MacMem$ is the biggest contributor to the runtime of our implementations, we explore various types of (keyed) cryptographic checksums and their performance on top of seL4. We compare the performance of five different MAC functions, namely, CBC-AES [85], HMAC-SHA-256 [5], Simon and Speck [73], and BLAKE2S [75], on 1MB of data in the user-space of seL4. The performance results in Figure 4.5a illustrate that the runtime of MAC based on Speck-64-128 [3] and BLAKE2S in seL4 are similar; and they are at least 33% faster than other MAC functions when running on Sabre Lite.

Performance of MacMem vs Memory Sizes. Another factor that affects $MacMem$'s performance is the size of memory regions to be attested. We experiment by creating another process in the user-space and perform attestation on various sizes (ranging from 1MB to 10MB) of memory regions inside that process. As expected, the results of this experiment, illustrated in Figure 4.5d, indicate that $MacMem$ performance is linear as a function of the attested memory sizes. This experiment also illustrates feasibility of performing attestation of 10MB of memory on top of seL4 in HYDRA using a Speck-based MAC in less than 250msec.

Performance on MacMem vs Numbers of Processes. This experiment answers the following question: How would an increase in number of processes affect the performance

[3]Speck with 64-bit block size and 128-bit key size

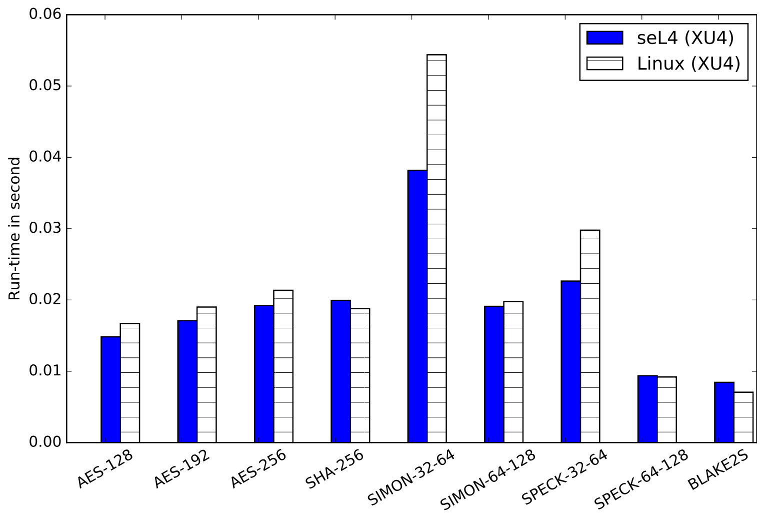


Figure 4.6: MAC implementations

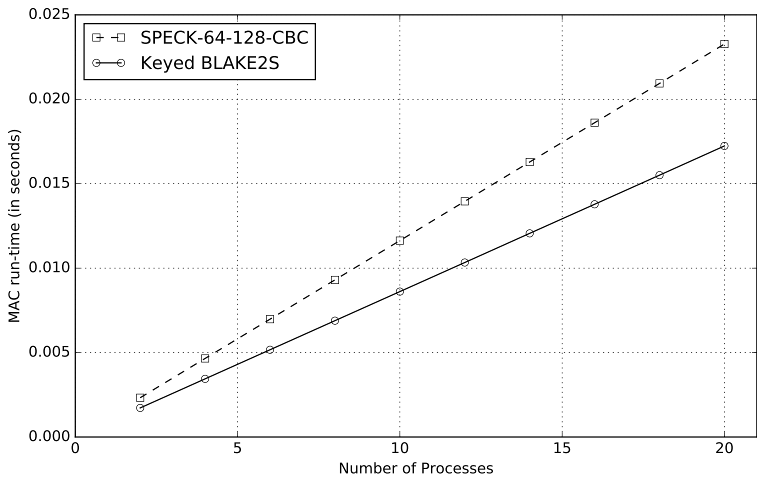


Figure 4.7: $MacMem$ vs num processes

of HYDRA? To answer it, we have the initial process spawn additional user-space processes (from 2 to 20 extra processes) and, then, perform $MacMem$ on 100 KB memory in each process. To ensure fair scheduling of every process, we set priority of all processes (including the initial process) to the maximum priority. The result from Figure 4.5e indicates that the performance of $MacMem$ is linear as a function of the number of processes on a Sabre Lite device.

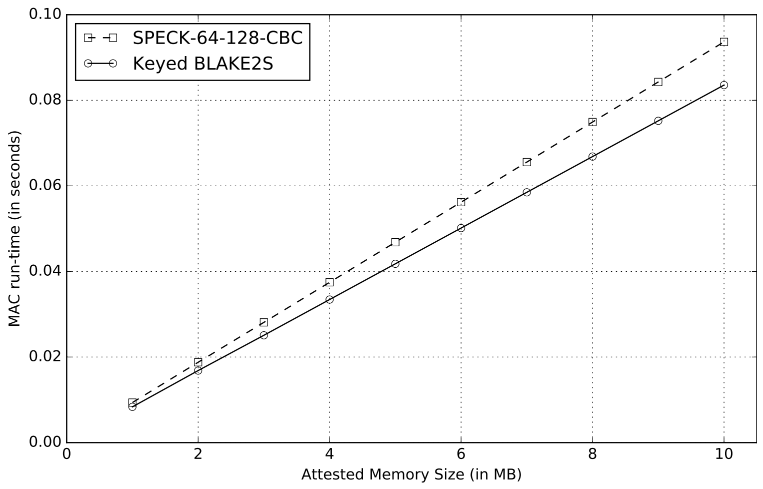


Figure 4.8: $MacMem$ vs mem size

4.5.2 Performance on ODROID-XU4

We also evaluate performance of HYDRA on ODROID-XU4 @ 2.1 GHz. Despite lacking an Ethernet driver, we evaluate the core component of HYDRA: $MacMem$. Unlike results from I.MX6 Sabre Lite, BLAKE2S-based MAC achieves the best performance for attesting 10MB on ODROID-XU4 platform.

MAC Performance on Linux vs in seL4. Figure 4.6 illustrates the performance comparison of keyed MAC functions on ODROID-XU4 running on Ubuntu 15.10 and seL4. Results support feasibility of $\mathcal{R}$A in seL4, since the runtime of seL4-based $\mathcal{R}$A can be as fast as that of $\mathcal{R}$A running on top of the popular Linux OS.

MAC Performance on ODROIX-XU4. Speck- and BLAKE2S-based MACs have the fastest attestation runtimes in seL4. We conducted additional experiments with these MAC functions on ODROID-XU4. Figure 4.7 shows the linear relationship between the number of processes and $MacMem$ runtime. Also, MAC runtime in Figure 4.8, is also linear in terms of the memory size to be attested. Finally, runtime of BLAKE2S-based MAC needs under 100 milliseconds to attest 10MB of memory.

Chapter 5

Shuffled Measurement

5.1 Introduction

Lower-end devices are often used in real-time and safety-critical applications. Thus, it is important to minimize the impact of security on normal operation (i.e., availability) of such devices. As motivated in Chapter 3, it is undesirable for a $\mathcal{R}\mathsf{A}$ technique to allow the measurement process ($\mathcal{M}\mathsf{P}$) on $\mathcal{P}\mathsf{rv}$ to run without interruption, considering that computing a measurement over a substantial amount of memory might take a relatively long time. In other words, $\mathcal{M}\mathsf{P}$ should be interruptible by a legitimate, time/safety-critical application. However, $\mathcal{P}\mathsf{rv}$ might have been compromised and the same safety-critical application might contain malware ($\mathcal{M}\mathsf{al}$). $\mathcal{M}\mathsf{al}$ presumably wants to evade detection. When confronted with $\mathcal{R}\mathsf{A}$, it may want to simply erase itself, perhaps in order to reappear later. Alternatively, it might remain on $\mathcal{P}\mathsf{rv}$ and try to avoid detection by relocating itself *during* $\mathcal{R}\mathsf{A}$. This behavior corresponds to *self-relocating malware*.

Therefore, this chapter focuses on reconciling two seemingly contradicting objectives: resistance against self-relocating malware and minimizing the impact of $\mathcal{R}\mathsf{A}$ on $\mathcal{P}\mathsf{rv}$'s availability.

In particular, we explore a mitigation technique that allows $\mathcal{M}\mathsf{P}$ to be interruptibly while protecting $\mathcal{P}\mathsf{rv}$ against self-relocating malware.

Prior hybrid $\mathcal{R}\mathsf{A}$ designs had different motivations. SMART avoids self-relocating $\mathcal{M}\mathsf{al}$ by enforcing non-interruptibility of the measuring process, which fully sacrifices interruptibility by a safety-critical task. Although TrustLite allows secure interrupts, it fails to detect self-relocating malware. Whereas, TyTAN protects against self-relocating malware by enforcing a rule that the process being measured can not interrupt $\mathcal{M}\mathsf{P}$, while other processes can. Nonetheless, this approach lacks transparency of $\mathcal{R}\mathsf{A}$, i.e., if a safety-critical process is being measured, it can not interrupt $\mathcal{M}\mathsf{P}$. Also, TyTAN might not detect self-relocating malware if process isolation is compromised (e.g., by a kernel bug) resulting in process collusion. Furthermore, TrustLite and (to a lesser degree) TyTAN require more advanced hardware features than SMART, which translates into extra cost for low-end platforms.

5.2 Remote Attestation via Shuffled Measurements

To mitigate the conflict between self-relocating malware ($\mathcal{M}\mathsf{al}$) detection and critical mission of $\mathcal{P}\mathsf{rv}$, we propose *SMARM*: a light-weight technique, based on shuffled measurements. *SMARM* allows $\mathcal{M}\mathsf{P}$ to be **interruptible**, while the order in which M is measured is determined randomly and privately, by $\mathcal{M}\mathsf{P}$. The rationale is that, if $\mathcal{M}\mathsf{al}$ remains unaware of what portions of M have been already measured (covered), it can not decide where to relocate itself to escape detection. $\mathcal{M}\mathsf{al}$'s optimal strategy (i.e., where and when to relocate) depends on its knowledge of the $\mathcal{R}\mathsf{A}$ coverage. However, based on reasonable assumptions about security of the $\mathcal{R}\mathsf{A}$ architecture, we show that $\mathcal{M}\mathsf{al}$ is detectable with significant probability.

Furthermore, since individual $\mathcal{R}\mathsf{A}$ instances are independent, the compound probability for

$\mathcal{M}$al to evade detection can be made negligible. As discussed in Section 5.6 below, although this increased level of security comes at the cost of running several measurements, no additional hardware features are needed. This results in **the first** low-cost and secure hybrid $\mathcal{R}$A technique that has no impact on $\mathcal{P}$rv's availability during $\mathcal{R}$A.

Shuffled (or random) memory coverage has been already suggested in the context of software-based $\mathcal{R}$A, However, it was done differently from *SMARM*, in several ways. First, random coverage of memory in software-based $\mathcal{R}$A is not secret, i.e., $\mathcal{M}$al is fully aware of the sequence of memory blocks traversal. In contrast, *SMARM* assumes secrecy of this traversal pattern (shuffling), since it is generated based on $\mathcal{V}$rf's one-time challenge and a secret key shared by $\mathcal{V}$rf and $\mathcal{P}$rv, which is inaccessible to $\mathcal{M}$al, as part of the underlying SMART architecture. Also, as described in [80, 79, 52], memory blocks are measured several times before all are processed at least once. This redundant coverage is likely due to size restrictions and non-optimizable constraints of $\mathcal{M}$P.

In the rest of this chapter, in the context of $\mathcal{P}$rv implementing *SMARM*, we analyze different evasion strategies based on self-relocating $\mathcal{M}$al's varying degree of knowledge about the progress of shuffled measurements.

5.3 Model and Assumptions

We assume that $\mathcal{P}$rv's memory is divided into n blocks $M_1, \ldots, M_n$. We require $\mathcal{P}$rv to conform to the SMART+ or HYDRA hybrid security architecture, as described in Chapter 2 and 4 respectively. In addition, *SMARM* entails one important change with respect to SMART+ and HYDRA:

> We relax the atomicity requirement of SMART+ and HYDRA such that the measurement process implemented by $\mathcal{M}$P can be interrupted after it measures each

memory block.

Let σ be a permutation randomly selected for a given $\mathcal{R}\mathsf{A}$ instance (measurement) of M. That is, block $M_{\sigma(i)}$ is measured at step i. Let $t_1, \ldots, t_n$ be the times at which blocks $M_{\sigma(1)}, \ldots, M_{\sigma(n)}$ are measured, respectively.

Let M^* be M in benign state. Let $R^* = F_K(M^*)$ be the measurement corresponding to the healthy state, computed by the measurement routine F using key K. Finally, let R be the measurement actually computed over M in a given $\mathcal{R}\mathsf{A}$ instance[1].

We define a *benign measurement* as the event $R = R^*$. We further define the probability of $\mathcal{M}\mathsf{al}$ evading detection with strategy S as:

$$P_S = \Pr(R = R^* \mid M \neq M^*)$$

We assume each M_i is measured under the same constraints as SMART+ or HYDRA. Namely, execution of $\mathcal{M}\mathsf{P}$ (over M_i) is atomic, access to K is protected, and memory is cleaned up (such that no traces of K remain) after the measurement of M_i is done.

Potential interruptions by other tasks running on $\mathcal{P}\mathsf{rv}$ must thus be scheduled between the measurement of two blocks. Let $t_{\max}$ denote the "maximum non-interruptibility interval". The size of a memory block M_i is set such that the time to measure it is at most $t_{\max}$.

5.4 Self-Relocating $\mathcal{M}\mathsf{al}$ Evasion Strategies

Since $\mathcal{M}\mathsf{al}$'s goal is to avoid detection, it must restore each block where it resides to a benign state before that block is measured. This section considers the optimal strategy for

[1]Since $\mathcal{M}\mathsf{P}$ takes a non-negligible amount of time and since M can change during that time, we can not write $R = F(M)$.

$\mathcal{M}\mathsf{al}$, under various assumptions about its capabilities and knowledge, and the associated probability of detection.

We initially assume that $\mathcal{M}\mathsf{al}$ occupies a single block. The case where it resides in multiple blocks is discussed later in Section 5.5.3. Without loss of generality, we also assume that $\mathcal{M}\mathsf{al}$ is active during the entire $\mathcal{M}\mathsf{P}$, and can thus interrupt it and make changes to any block of M at any point as long as it does so between the intervals of $\mathcal{M}\mathsf{P}$ measuring a single block. Reactive $\mathcal{M}\mathsf{al}$, and $\mathcal{M}\mathsf{al}$ with restricted number of interruptions are discussed in Sections 5.5.4 and 5.5.5, respectively.

5.4.1 Erasure

One trivial evasion strategy for $\mathcal{M}\mathsf{al}$ is to simply erase itself as soon as possible, perhaps to re-infect $\mathcal{P}\mathsf{rv}$ at a later time. Assuming that $\mathcal{M}\mathsf{al}$ is aware of the incoming attestation request from $\mathcal{V}\mathsf{rf}$, or it interrupts $\mathcal{M}\mathsf{P}$ before it starts (or early on during its execution), erasure seems difficult, if not impossible, to mitigate. In the rest of this paper, we focus on strategies whereby $\mathcal{M}\mathsf{al}$ attempts to remain on $\mathcal{P}\mathsf{rv}$ while evading detection.

5.4.2 Relocation Techniques

Clearly, if $\mathcal{M}\mathsf{al}$ remains where it is, it can not escape detection. Otherwise, it must relocate itself, at least once. We identify and explore three intuitive $\mathcal{M}\mathsf{al}$ flavors (which vary in the degree of knowledge) and their associated probabilities of successful evasion. As mentioned earlier, we assume below that $\mathcal{M}\mathsf{al}$ occupies a single memory block.

Knowledge of Future Volume (KFV)

During $\mathcal{R}\mathsf{A}$, the volume (size) of memory that has not yet been measured is the least amount of actionable information that $\mathcal{M}\mathsf{al}$ might have. This knowledge can be acquired by measuring the time elapsed since the start of $\mathcal{M}\mathsf{P}$ and estimating the number of memory blocks already measured. This is based on a realistic assumption that $\mathcal{M}\mathsf{al}$ is aware of: (1) time when $\mathcal{M}\mathsf{P}$ began, and (2) time to measure one memory block. We refer to this degree of knowledge as the KFV model.

Theorem 5.1. *The optimal strategy for KFV* $\mathcal{M}\mathsf{al}$ *is to relocate after every memory block is measured. It would thus move a total of* $n-1$ *times, assuming that it can interrupt* $\mathcal{M}\mathsf{P}$ *at each block boundary. The probability of evasion is:*

$$P_{FV} = \left(1 - \frac{1}{n}\right)^n \approx e^{-1} \approx 0.37$$

Proof. Let M_{m_i} denote the block containing $\mathcal{M}\mathsf{al}$ at t_i, for $1 \leq m_i \leq n$. Having the ability to interrupt $\mathcal{M}\mathsf{P}$ between t_i and t_{i+1}, $\mathcal{M}\mathsf{al}$ can either stay put or relocate. Let p_i be the probability of $\mathcal{M}\mathsf{al}$ getting "caught" exactly at t_{i+1}:

$$p_i = \Pr(m_{i+1} = \sigma(i+1) \mid m_k \neq \sigma(k), 1 \leq k < i),$$

for $1 \leq i < n$.

If $\mathcal{M}\mathsf{al}$ relocates, two outcomes may occur: either (1) its new location $M_{m_{i+1}}$ has been already measured (there are i such blocks), in which case it will certainly not be caught, or (2) $M_{m_{i+1}}$ was not measured yet (there are $n-i$ such locations), in which case it will be caught with probability $\frac{1}{n-i}$. Consequently:

$$p_i^{\text{move}} = \frac{i}{n} \cdot 0 + \frac{n-i}{n} \cdot \frac{1}{n-i} = \frac{1}{n}. \tag{5.1}$$

If $\mathcal{M}\mathsf{al}$ does not relocate, two situations may occur. Let j be the last interval when $\mathcal{M}\mathsf{al}$ moved. (If it never moved, $j = 0$.) Again, $M_{m_{i+1}} = M_{m_{j+1}}$ might have been already measured, in which case $\mathcal{M}\mathsf{al}$ will not be caught. This occurs with probability $\frac{j}{n}$. Indeed, since $M_{m_{j+1}}$ can not have been measured in the last $i - j$ steps (since we assume $\mathcal{M}\mathsf{al}$ has not been detected so far), it must have been measured in the j first ones.

Otherwise, if $M_{m_{j+1}}$ has not been measured yet (which occurs with probability $1 - \frac{j}{n}$), $\mathcal{M}\mathsf{al}$ will be caught with probability $\frac{1}{n-i}$ (for the same reason as in Eq. (5.1)). Consequently,

$$p_i^{\text{stay}} = \frac{j}{n} \cdot 0 + \frac{n-j}{n} \cdot \frac{1}{n-i} = \frac{n-j}{n(n-i)}$$

If $j < i$, $p_i^{\text{stay}} > p_i^{\text{move}}$. Therefore, it is always beneficial for $\mathcal{M}\mathsf{al}$ to relocate. Interestingly, the new location $M_{m_{i+1}}$ does not matter, as long as $m_{i+1} \neq m_i$. Relocation at each interval leads to an overall evasion probability of:

$$P_{\text{FV}} = (1 - p_i)^{n-1} \cdot \Pr(m_1 \neq \sigma(1)) = \left(1 - \frac{1}{n}\right)^n$$

$\Pr(m_1 \neq \sigma(1))$ is the probability that the first block measured is not M_{m_1}. The approximation to e^{-1} results from the limit definition of e. □

Knowledge of Future Coverage (KFC)

In addition to knowing *how many* blocks have been measured, $\mathcal{M}\mathsf{al}$ might also learn *which* blocks have been measured. That is, after t_i, $\mathcal{M}\mathsf{al}$ knows $\{\sigma(1), \ldots, \sigma(i)\}$. Based on this, $\mathcal{M}\mathsf{al}$ can infer $\{\sigma(i+1), \ldots, \sigma(n)\}$. This is different from knowing precise values $\sigma(i+1), \ldots, \sigma(n)$; see Section 5.4.2 below. We refer to this as the KFC model.

This greater knowledge might stem from a side-channel vulnerability in the implementation of $\mathcal{M}\mathsf{P}$, e.g., if measured blocks can be distinguished from non-measured ones from the point of view of $\mathcal{M}\mathsf{al}$. Alternatively, it could be the result of $\mathcal{M}\mathsf{P}$ carelessly storing $\{\sigma(1), \ldots, \sigma(i)\}$ (See Section 5.7).

Theorem 5.2. *The optimal strategy for KFC $\mathcal{M}\mathsf{al}$ is to change its location to $M_{\sigma(1)}$ immediately after t_1. The probability of evasion is:*

$$P_{FC} = 1 - \frac{1}{n}$$

Proof. Since $\sigma(1)$ is unknown before t_1, $\mathcal{M}\mathsf{al}$ can not make any informed decision, and thus would be discovered if $m_1 = \sigma(1)$. After t_1, $\mathcal{M}\mathsf{al}$ knows that $M_{\sigma(1)}$ was measured. Thus, it can safely relocate there and remain until the end. The result follows trivially: $\Pr(m_1 = \sigma(1)) = \frac{1}{n}$. □

Knowledge of Future Order (KFO)

In addition to future coverage, $\mathcal{M}\mathsf{al}$ might know $\sigma(i+1), \ldots, \sigma(n)$, and thus also $\{\sigma(1), \ldots, \sigma(i)\}$. Whether $\mathcal{M}\mathsf{al}$ also knows the past order of coverage $(\sigma(1), \ldots, \sigma(i))$ is irrelevant; see Section 5.5. $\mathcal{M}\mathsf{al}$ might acquire this additional knowledge due to $\sigma(i+1), \ldots, \sigma(n)$ being insecurely stored by $\mathcal{M}\mathsf{P}$; see Section 5.5. The leakage can also occur due to some vulnerability in the random block selection process, e.g., in case of a weak random number generation.

We refer to this as the KFO model.

Theorem 5.3. *The optimal strategy for KFO $\mathcal{M}\mathsf{al}$ is to relocate to any M_j with $j \neq \sigma(1)$ before t_1 if $M_{\sigma(1)}$ is its initial location, and then to $M_{\sigma(1)}$ immediately after t_1. The probability of evasion is:*

$$P_{FO} = 1$$

Proof. If $\mathcal{M}\mathsf{al}$ is located in $\sigma(1)$ (the next block to be measured), it must relocate. After t_1, $\mathcal{M}\mathsf{al}$ knows that $M_{\sigma(1)}$ was measured, so it can safely relocate there and remain to the end. Consequently, $\mathcal{M}\mathsf{al}$ must move once (or twice, if $M_{\sigma(1)}$ is its initial location) in order to evade detection. □

5.5 Discussion

We now discuss the viability of aforementioned malware flavors and several related issues.

5.5.1 Realistic Scenarios

We believe that KFV is quite realistic. It requires little insight and sophistication from $\mathcal{M}\mathsf{al}$. It is viable even with a secure hybrid $\mathcal{R}\mathsf{A}$ framework and a secure handling of σ. As discussed in Section 5.5.2, even when $\mathcal{M}\mathsf{al}$'s timing estimates are inaccurate, it can attain the probability of evasion close to e^{-1}. In contrast, KFC and KFO $\mathcal{M}\mathsf{al}$ flavors are more far-fetched, as long as σ is placed in secure storage, or stored elsewhere in an encrypted and authenticated form; see Section 5.7.[2] Therefore, the rest of this paper focuses on KFV $\mathcal{M}\mathsf{al}$.

[2]However, KFC or KFO might be possible if faulty $\mathcal{M}\mathsf{P}$ leaks information about σ, or if σ is leaked via some other side-channel means.

5.5.2 Unknown Timing

Recall that KFV $\mathcal{M}\mathsf{al}$ is assumed to know $(t_1, \dots, t_n)$. This may not always be the case in practice. For instance, other active processes might distort the timing, or spurious delays might be introduced by the $\mathcal{M}\mathsf{P}$, as a potential countermeasure.

One possible strategy for $\mathcal{M}\mathsf{al}$ is to relocate as often as possible. As follows from the proof of Theorem 5.1, as long as $\mathcal{M}\mathsf{al}$ relocates at least once between two blocks being measured, the predicted probability of successful evasion is reached. While precise values for $(t_1, \dots, t_n)$ may not be known, $\mathcal{M}\mathsf{al}$ can easily determine the lower bound on the time to measure one block. If $\mathcal{M}\mathsf{al}$ relocates with a frequency of that lower bound, it can attain the evasion probability of e^{-1}.

5.5.3 $\mathcal{M}\mathsf{al}$ in Multiple Blocks

Despite the earlier assumption, $\mathcal{M}\mathsf{al}$ might not fit in a single memory block. Let s denote the number of blocks occupied by KFV $\mathcal{M}\mathsf{al}$. Its evasion probability can be viewed as evasion by s separate single-block pieces of malware, with the probability of:

$$P_{\text{FV}}(S = s) = P_{\text{FV}}(S = 1)^s \approx e^{-s}.$$

5.5.4 Active *vs* Reactive Malware

So far, we assumed that $\mathcal{M}\mathsf{al}$ is active throughout $\mathcal{M}\mathsf{P}$. We now broaden this to include $\mathcal{M}\mathsf{al}$ which wakes up (becomes active) at some point during $\mathcal{R}\mathsf{A}$. Let w be the number of memory blocks measured thus far. Let $P_{\text{active}}(n, s)$ be the probability of evasion of an active $\mathcal{M}\mathsf{al}$ of size s for a given strategy on measurement of n blocks. The probability $P_{\text{reactive}}(n, s, w)$ of

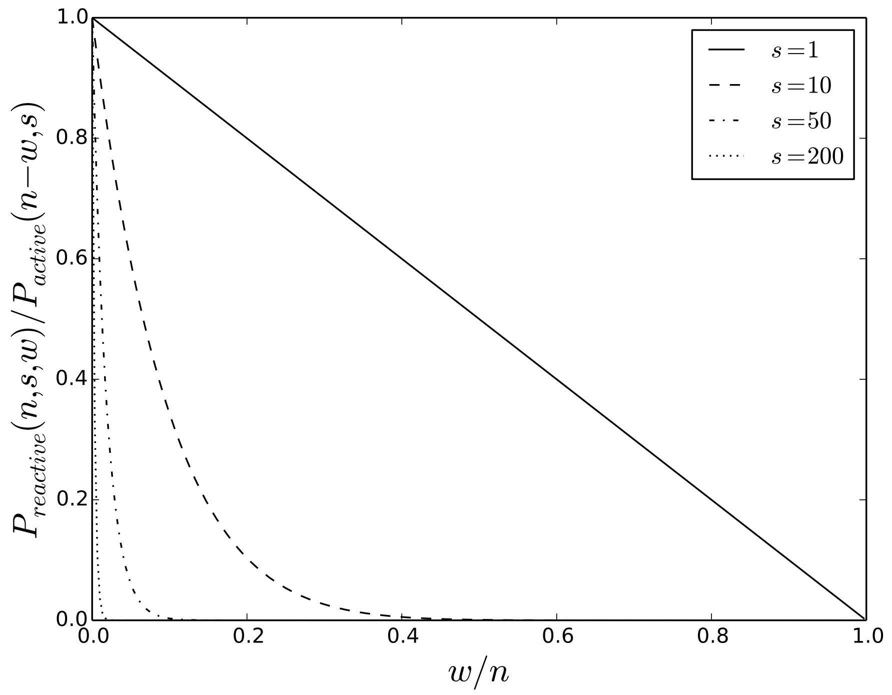


Figure 5.1: Effect of reactive malware.

$\mathcal{M}\mathsf{al}$ following the same strategy reacting after w blocks is:

Theorem 5.4.

$$P_{reactive}(n,s,w) = \frac{\binom{n-s}{w}}{\binom{n}{w}} P_{active}(n-w,s)$$

Proof. The $\frac{\binom{n-s}{w}}{\binom{n}{w}}$ factor is the probability of none of $\mathcal{M}\mathsf{al}$'s s blocks being chosen in first w measurements. It is a special case of a hypergeometric distribution. The second factor is the probability that an active $\mathcal{M}\mathsf{al}$ escapes detection, with $n-w$ blocks remaining. □

It follows trivially that $P_{\text{reactive}}(n,s,0) = P_{\text{active}}(n,s)$. Figure 5.1 shows the effect of $\mathcal{M}\mathsf{al}$ that is only active after w steps.

5.5.5 Limited # of Interruptions

We now consider the case when KFV $\mathcal{M}\mathsf{al}$ is limited to at most k interruptions during the entire $\mathcal{M}\mathsf{P}$ instance. This might be motivated by $\mathcal{M}\mathsf{al}$ aiming to reduce its timing footprint. Indeed, one possible countermeasure for $\mathcal{M}\mathsf{P}$ (with *SMARM*) that suspects self-relocating $\mathcal{M}\mathsf{al}$ presence is to measure its total elapsed (wall-clock) time. If it diverges significantly from the expected time, and if there is no legitimate justification, it can be an indication of activity by frequently relocating $\mathcal{M}\mathsf{al}$.

Theorem 5.5. *The optimal strategy for KFV $\mathcal{M}\mathsf{al}$ limited to k interruptions is to change its location after each group of $n/(k+1)$ blocks is measured. It thus moves k times and its probability of evasion is:*

$$P_{FV}(K = k) = \left(1 - \frac{1}{k+1}\right)^{k+1}$$

Proof. Theorem 5.1 showed that relocating is always preferable for $\mathcal{M}\mathsf{al}$ to remaining in the same block. What remains to determine is when to optimally make theses interruptions.

Let $n_1, \ldots, n_k$ be the number of blocks measured before each interruption, and let $n_{k+1} = n - \sum_{i=1}^{k} n_i$ be the remaining blocks after the last interruption. The probability for $\mathcal{M}\mathsf{al}$ to escape detection is thus:

$$P = \prod_{i=1}^{k+1} \left(1 - \frac{n_i}{n}\right)$$

Let P be the probability pertaining to the strategy $n_1, \ldots, n_k$. Let an alternative strategy

$n'_1, \ldots, n'_k$ (and the corresponding probability P') be defined as:

$$n'_i = \begin{cases} n_a + c \text{ if } i = a \\ n_b - c \text{ if } i = b \\ n_i \text{ else,} \end{cases}$$

for $a \neq b : n_a \geq n/(k+1), n_b \leq n/(k+1)$. Such a pair is guaranteed to exist because $n_{k+1} = n - \sum_{i=1}^{k} n_i$. We thus have:

$$\begin{aligned} \frac{P}{P'} &= \frac{\left(1 - \frac{n_a}{n}\right)\left(1 - \frac{n_b}{n}\right)}{\left(1 - \frac{n_a+c}{n}\right)\left(1 - \frac{n_b-c}{n}\right)} \\ &= \frac{\left(1 - \frac{n_a}{n}\right)\left(1 - \frac{n_b}{n}\right)}{\left(1 - \frac{n_a}{n}\right)\left(1 - \frac{n_b}{N}\right) + \frac{c}{n}\left(\frac{n_b}{n} - \frac{n_a}{n}\right) - \frac{c^2}{n^2}}. \end{aligned}$$

Since $n_a \geq n_b$, the denominator is smaller than the numerator, and thus $P \geq P'$. This shows that any strategy that diverges from $n_i = n/(k+1)$ will be sub-optimal.

$\square$

5.6 Reliable Detection

Using *SMARM*, the probability that one measurement does not detect $\mathcal{M}\mathsf{al}$ presence is non-negligible. However, if multiple measurements are taken, the overall false negative probability decreases exponentially. Multiple measurements can be obtained via: (1) independent consecutive $\mathcal{R}\mathsf{A}$ instances, (2) batch mode, e.g. $\mathcal{V}\mathsf{rf}$ sends m challenges at once and receives m measurements, or (3) self-measurements by $\mathcal{P}\mathsf{rv}$ itself, as described in [20]. Given m independent measurements, each with probability P of a false negative, the overall false negative probability is $P_m = P^m$. That is, with $P = e^{-1}$ this gives, e.g., $P_7 < 10^{-3}$, and $P_{13} < 10^{-6}$.

5.7 Block Permutation in Practice

This section discusses a trial implementation of random block shuffling and its security implications.

5.7.1 Permutation Computation and Storage

Recall that the random one-time permutation σ is computed at the start of $\mathcal{M}\mathsf{P}$, prior to measuring any memory blocks. One way to compute it efficiently is by using the well-known *Fisher-Yates* [32] (also known as *Knuth*'s [46]) shuffle method.

$\mathcal{P}\mathsf{rv}$'s PRNG is seeded with $H(c, K)$ where c is $\mathcal{V}\mathsf{rf}$'s challenge, K is the key securely stored by $\mathcal{P}\mathsf{rv}$ (as part of SMART), and H is a hash function. As mentioned in Section 6.3, we assume that $\mathcal{P}\mathsf{rv}$ authenticates each attestation request. Once an attestation request is validated, $\mathcal{P}\mathsf{rv}$ knows that c (contained therein) is unique; hence, $H(c, K)$ is unique as well. This guarantees that random values σ_i produced by PRNG are both fresh and secret, i.e., unpredictable by $\mathcal{M}\mathsf{al}$.

Once computed, $\sigma = \{\sigma(1), \ldots, \sigma(n)\}$ is stored in secure memory. The underlying security architecture ensures that (similar to K), σ can be written and read only by $\mathcal{M}\mathsf{P}$. This requires $n\lceil \log n \rceil$ bits of storage. When the measurement of block $M_{\sigma(i-1)}$ is completed (or when $\mathcal{M}\mathsf{P}$ starts at $i = 1$), $\sigma(i)$ is read and $M_{\sigma(i)}$ is fed to the MAC function. Once all blocks are measured, σ remains in secure storage. It is then over-written at the start of the next $\mathcal{R}\mathsf{A}$ instance.

If the size of the additional secure storage is a concern, the following variant is an alternative. The permutation σ is instead stored encrypted and authenticated in *insecure storage*, and no additional secure storage is required. Each index $\sigma(i)$ is encrypted individually as a block of the block cipher. Since efficiency is a concern, the block cipher operates in CTR mode [26], so that random access to individual $\sigma(i)$ is possible. The counter is set to a concatenation of the $\mathcal{V}\mathsf{rf}$ one-time challenge c and i. The encryption of $\sigma(i)$ is thus $E_K(c||i) \oplus \sigma(i)$.

In addition, a MAC of the encrypted $\sigma(i)$ is concatenated (cf. Encrypt-then-Mac [40]). A scenario where $n > 2^{128}$ being unrealistic, using a block cipher and MAC with a block size of 128 is sufficient. For instance, using AES-128 as E and a SHA-2-based HMAC means that σ is stored on $48n$ bytes. At the start of the measurement of each block, the encrypted $\sigma(i)$ is decrypted, its MAC verified, and the corresponding block read.

This variant trades off secure storage for regular storage and increased computation cost (see experiment results in Section 5.8).

5.7.2 Memory Overhead

We estimate memory overhead for storing σ on I.MX6-SabreLite [10], a popular and inexpensive development board representative of low- to medium-end IoT devices.

As discussed in Section 6.3, the maximum non-interruptibility interval $t_{\max}$ influences block size and their number. Given total memory size $|M|$ and throughput θ $n = \frac{|M|}{\theta t_{\max}}$. This allows us to predict the amount of storage for σ. Figure 5.2 shows this overhead, for varying values of $t_{\max}$ and $|M|$, with a throughput of $\theta \approx 20.48$MB/s.

Figure 5.3 shows the probability of evasion for KFV $\mathcal{M}\mathsf{al}$. It shows that a higher $t_{\max}$ decreases the chances of escaping detection. As evident from Figure 5.2, it also requires less storage. However, in order to guarantee good availability on $\mathcal{P}\mathsf{rv}$, $t_{\max}$ should not be too high.

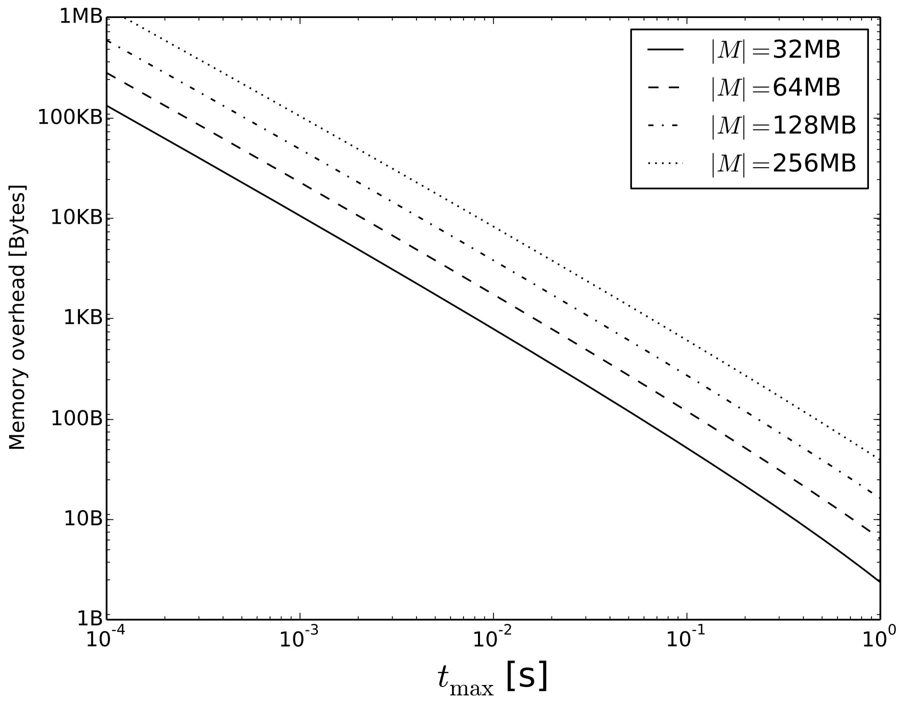


Figure 5.2: Memory requirement for σ on a I.MX6-SabreLite, as a factor of $t_{\max}$ and $|M|$.

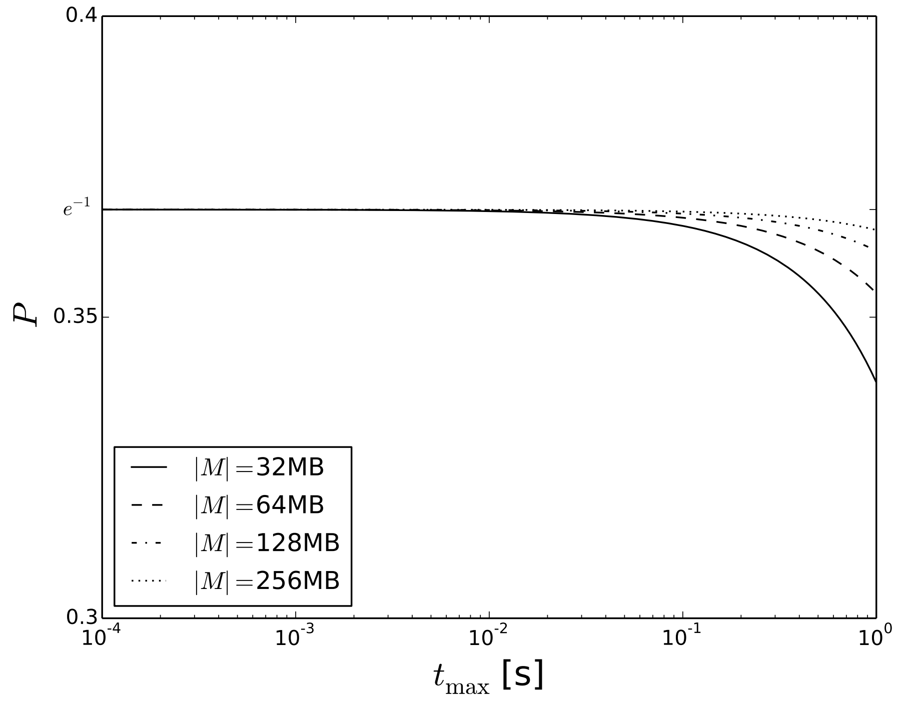


Figure 5.3: Probability of evasion for KFV $\mathcal{M}$al, on a I.MX6-SabreLite, as a factor of $t_{\max}$ and $|M|$. $\mathcal{M}$al is proactive and is of size $s = 1$.

5.8 HYDRA Implementation

5.8.1 Overview

We implemented *SMARM* in the context of HYDRA [30] on an I.MX6-SabreLite. Recall from Chapter 4 that HYDRA implements a hybrid $\mathcal{R}\mathsf{A}$ design for devices with a Memory Management Unit (MMU). It builds upon the formally verified seL4 [45] 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.

HYDRA stores K and attestation code in a writable memory region and configures the system such that no other process, besides $\mathcal{M}\mathsf{P}$, can access these memory regions. Access control configuration in HYDRA also involves $\mathcal{M}\mathsf{P}$ having exclusive access to its thread control block as well as to memory regions used for key-related computations. $\mathcal{M}\mathsf{P}$ starts with the highest maximum controlled priority (MCP). This ensures $\mathcal{M}\mathsf{P}$ execution will run uninterruptedly as long as its "priority" remains the highest. Note that, in seL4, a priority is the effective priority of a process, which can be increased and decreased at run-time as long as it does not exceed its MCP value. In contrast, a process cannot increase its MCP after it is set (but a decrease is possible).

The formally verified version of seL4 uses a simple preemptive round-robin scheduler. Processes with the same priority take turns to execute for the same time-slice of length t_{slice}, unless pre-empted by a higher-priority process.

The measurement procedure in the HYDRA-based *SMARM* implementation is as follows. At the start of a time-slice, $\mathcal{M}\mathsf{P}$ is set to highest priority (so that it can not be interrupted). Once the measurement of the block is done (this takes $t_{\max}$ seconds), $\mathcal{M}\mathsf{P}$ decreases its priority and yields (via the `seL4_Yield` system call) its remainder of the time-slice. This

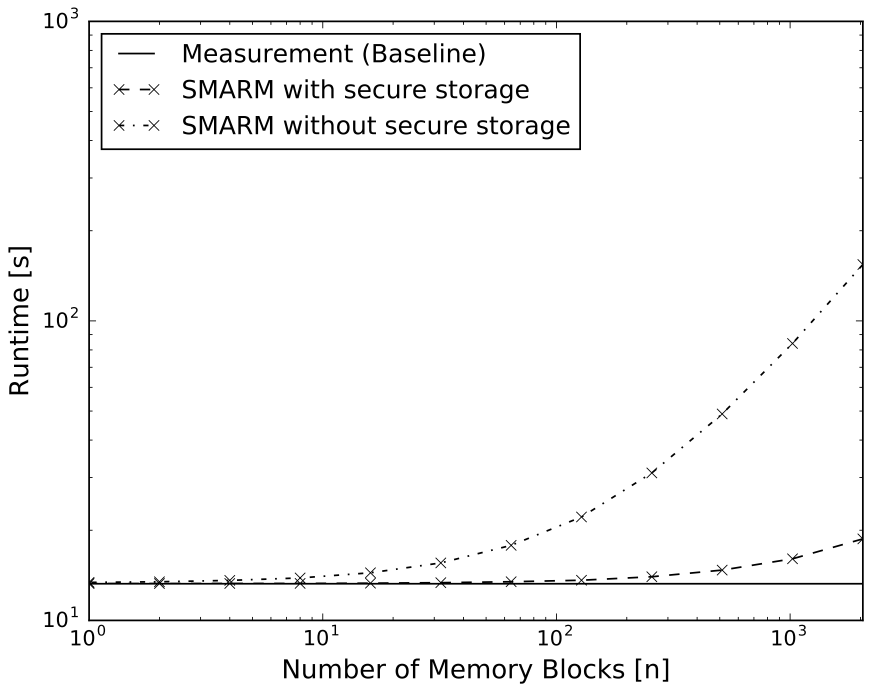


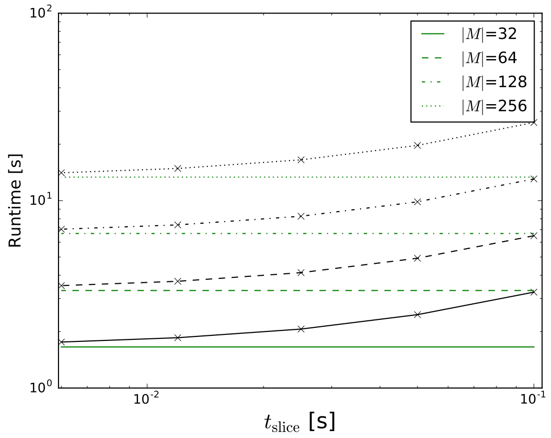
Figure 5.4: Runtime of *SMARM* with/without secure storage for σ as a factor of n. $|M|$ is fixed to 256MB. It is also assumed that no other process exists besides $\mathcal{M}\mathsf{P}$.

allows another process (which may include $\mathcal{M}\mathsf{al}$) to run immediately after each block is measured with a fresh time-slice. Once the next time-slice of $\mathcal{M}\mathsf{P}$ begins, the next block is measured [3].

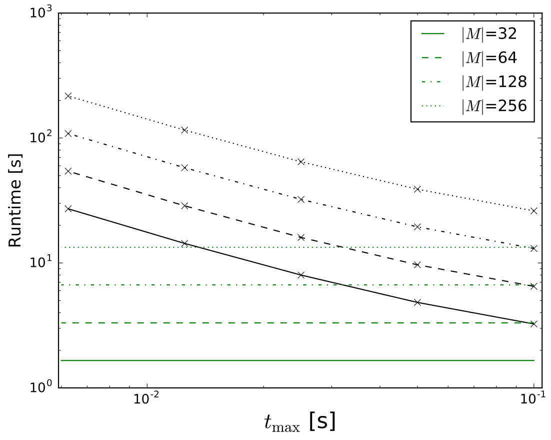
5.8.2 Experimental Results: SMARM with/without Secure Storage

We generate the random permutation σ using the Fisher-Yates shuffle method, discussed in Section 5.7. The PRNG is implemented using AES-256 in CTR Mode, seeded with SHA-256(c, K). In the variant where secure storage is not available, we implement the underlying block cipher and MAC function as AES-128 and SHA-256-based HMAC respectively.

[3]Technically, each interrupting process, besides $\mathcal{M}\mathsf{P}$, is allowed one time-slice (round-robin scheduling). However, for simplicity we assume a single (possibly infected) such process performing $\mathcal{P}\mathsf{rv}$'s main task.



(a) Runtime as a factor of t_{slice} and $|M|$. $t_{\max}$ is fixed to 100ms.



(b) Runtime as a factor of $t_{\max}$ and $|M|$. t_{slice} is fixed to 100ms.

Figure 5.5: Runtime comparison between *SMARM* (crosses, black) and baseline measurement (green) with different t_{slice}, $t_{\max}$ and $|M|$; it is assumed that a single process exists besides $\mathcal{M}$P.

Figure 5.4 shows overall runtime of these two variants (i.e. *SMARM* with/without secure permutation storage) compared to that of the baseline measurement. With enough secure storage, *SMARM* incurs 41% overhead over the baseline measurement with $n = 2048$ (or $t_{\max} = 0.006$s). This overhead decreases as n decreases; it achieves 0.8% at $n = 32$ (or $t_{\max} = 0.4$s). On the other hand, without secure storage for σ, *SMARM* spends significant amount of time performing additional encryptions and MAC computations. The overhead in this case can be as high as 10,636% (at $n = 2048$) and as low as 2% (at $n = 1$). Thus, it might be more beneficial to deploy a lower number of memory blocks if a device does not have room for secure storage.

5.8.3 Experimental Results: Different t_{slice} and $t_{\max}$

Figure 5.5 illustrates the worst-case runtime of *SMARM* on this HYDRA implementation, compared to the uninterrupted measurement of M, as a factor of t_{slice}, $t_{\max}$, and $|M|$. The overall *SMARM* runtime: (1) increases as t_{slice} increases, and (2) decreases as $t_{\max}$ increases. We found that the overhead can be up to a factor of $1 + \frac{t_{\text{slice}}}{t_{\max}}$, while another process can run for up to t_{slice} seconds every $t_{\text{slice}} + t_{\max}$ seconds.

Chapter 6

Memory Locking

6.1 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 safety-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-*

[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.

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 $\mathcal{R}\mathsf{A}$. $\mathcal{R}\mathsf{A}$ is a security service for remotely assessing integrity of software and memory (as well as other types of storage) in embedded devices. It is typically realized as an interaction between a trusted entity (*verifier* or $\mathcal{V}\mathsf{rf}$) and an untrusted, potentially malware-infected, remote device (*prover* or $\mathcal{P}\mathsf{rv}$). 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 $\mathcal{R}\mathsf{A}$) to prevent malware from falsifying results. Consistency is of paramount concern for $\mathcal{R}\mathsf{A}$, 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 $\mathcal{R}\mathsf{A}$ 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?
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

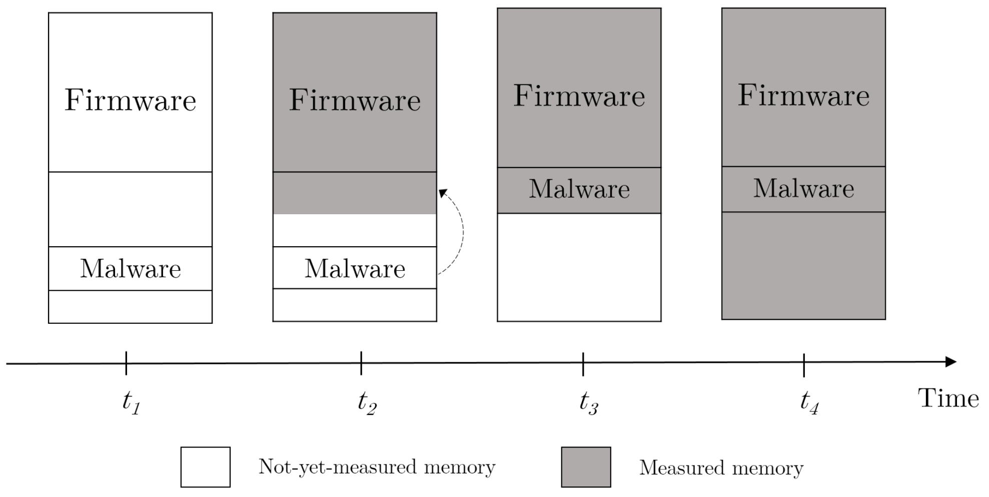


Figure 6.1: Program memory of infected $\mathcal{P}\mathsf{rv}$ before (at t_1), during (at t_2, t_3) and after (at t_4) the measurement process.

its safety-critical function, software integrity of this device is periodically checked using $\mathcal{R}\mathsf{A}$. 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 safety-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 *self-relocating 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. Figure 6.1 illustrates how such malware can escape detection during execution of the measurement process ($\mathcal{M}\mathsf{P}$). The attack can be constructed as follows:

1. At time t_0, malware enters and infects $\mathcal{P}\mathsf{rv}$. We assume that malware resides at the tail end of program memory. If program memory is insufficient to contain both existing firmware and malware, the latter can use the executable compression technique [90] to reduce the sizes of both firmware and itself.

2. At time $t_1 > t_0$, malware intercepts $\mathcal{V}\mathsf{rf}$'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 $\mathcal{M}\mathsf{P}$.

3. $\mathcal{M}\mathsf{P}$ runs without interruption from t_1 to t_2.

4. At $t_2 > t_1$, malware interrupts $\mathcal{M}\mathsf{P}$. It then copies itself to the part of memory that was already measured, erases itself from its prior location, and resumes execution of $\mathcal{M}\mathsf{P}$.

5. At time t_4, $\mathcal{M}\mathsf{P}$ completes and produces the measurement for delivery to $\mathcal{V}\mathsf{rf}$.

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

Although dangers of self-relocating malware were anticipated in the design of some software-based attestation methods, e.g., Viper and Pioneer [52, 79], 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 self-relocating malware, since their measurement process involves precise timing which would be noticeably skewed by self-relocating malware (due to the latter's efforts of copying and erasing). However, they are also unsuitable for *remote* attestation where fluctuating network delays influence overall timing. Thus, the main goal of this chapter 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.

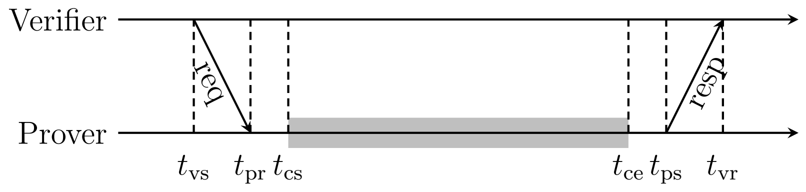


Figure 6.2: Timeline for a typical $\mathcal{R}\mathsf{A}$ scheme. Verifier's request is sent at t_{vs} and received at t_{pr}. Computation starts at t_{cs} and ends at t_{ce}. Report is sent at t_{ps} and received at t_{vr}.

6.2 Temporal Consistency

In recent years, $\mathcal{R}\mathsf{A}$ emerged as a distinct security service for detecting malware on CPS, ES and IoT devices. It involves verification of current internal state (i.e., RAM or flash) of an untrusted remote hardware platform (prover or $\mathcal{P}\mathsf{rv}$) by a trusted entity (verifier or $\mathcal{V}\mathsf{rf}$). $\mathcal{R}\mathsf{A}$ can help the latter establish a static or dynamic root of trust in $\mathcal{P}\mathsf{rv}$ and can also be used to construct other security services, such as software updates [78] and secure deletion [69].

6.2.1 $\mathcal{R}\mathsf{A}$ Blueprint

A typical $\mathcal{R}\mathsf{A}$ scheme operates as follows:

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

The timeline for this sequence of events is shown in Figure 6.2. Computation of $\mathcal{M}\mathsf{P}$ (in gray) may be deferred due to networking delays, $\mathcal{V}\mathsf{rf}$'s request authentication, or termination of the previously running task. However, typically, $t_{\mathrm{pr}} \approx t_{\mathrm{cs}}$ and $t_{\mathrm{ce}} \approx t_{\mathrm{ps}}$. Also, $\mathcal{P}\mathsf{rv}$ has no control over t_{vs} and t_{vr}. Consequently, hereafter we only consider $t_{\mathrm{s}} := t_{\mathrm{cs}}$ (with $t_{\mathrm{cs}} = t_{\mathrm{pr}}$) and $t_{\mathrm{e}} := t_{\mathrm{ce}}$ (with $t_{\mathrm{ps}} = t_{\mathrm{ce}}$).

As discussed in Section 6.1, $\mathcal{M}\mathsf{P}$ may require time-consuming computations. The exact time it takes depends on the size of $\mathcal{P}\mathsf{rv}$'s memory, its computational capability, and the underlying cryptographic function(s). As a sample hardware platform, we consider $\mathcal{M}\mathsf{P}$ running an ODROID-XU4 [37] – 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 6.3 shows the costs of these operations, for various attested memory sizes and cryptographic algorithms[3]. Above 1MB, $\mathcal{M}\mathsf{P}$ 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 $\mathcal{M}\mathsf{P}$ to run uninterrupted.

Recent hybrid $\mathcal{R}\mathsf{A}$ architectures, such as TrustLite [47] and TyTAN [11], permit tasks to be interrupted. While this allows for time-/safety-critical processes to run and preserve $\mathcal{P}\mathsf{rv}$'s critical functionality, attestation results might be *inconsistent*. Indeed, in TrustLite, since memory can change *during* execution of $\mathcal{M}\mathsf{P}$, the report produced and sent to $\mathcal{V}\mathsf{rf}$ might correspond to a state of $\mathcal{P}\mathsf{rv}$'s memory that *never existed in its entirety* at any given time. This is problematic if $\mathcal{P}\mathsf{rv}$ is infected with self-relocating malware. Assuming that such malware resides in the second half of $\mathcal{P}\mathsf{rv}$'s memory, it can interrupt $\mathcal{M}\mathsf{P}$ after the latter covers the first half of $\mathcal{P}\mathsf{rv}$'s memory, copy itself into the first half, erase traces in its former location, and resume $\mathcal{M}\mathsf{P}$. This way, malware remains undetected despite the fact that all

[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.

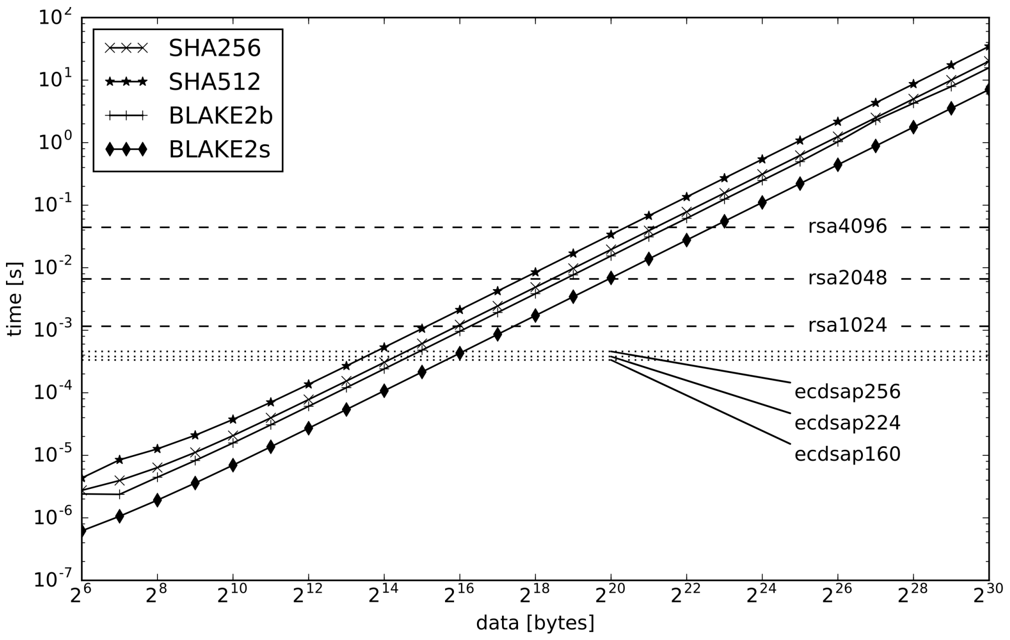


Figure 6.3: Computational costs of several hash functions and digital signatures on ODROID-XU4.

memory locations have been measured.

In TyTAN [11], 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 [31] disables interrupts as the first step in $\mathcal{M}$P. This precludes self-relocating 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.

6.2.2 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 $\mathcal{M}\mathsf{P}$ has exclusive write access. This way, computation can be performed on the copy and $\mathcal{M}\mathsf{P}$ 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 $\mathcal{M}\mathsf{P}$ 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, self-relocating malware can, in principle, still evade detection. This is further discussed in Section 6.4.3.

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

6.3 Modeling Temporal Consistency

We now introduce the model and notation for temporal consistency and supporting mechanisms. Although we focus on $\mathcal{R}\mathsf{A}$, the model is generic and relevant to other application domains that involve integrity-ensuring functions.

We assume that input data is located in $\mathcal{P}\mathsf{rv}$'s memory M, which consists of n contiguous blocks $[M_1 \dots\ M_n]$. 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 6.1. *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 $\mathcal{R}\mathsf{A}$, this holds since $\mathcal{M}\mathsf{P}$ (containing F) is protected by the underlying security architecture. For example, in hybrid attestation architectures, such as TrustLite, TyTAN and SMART, $\mathcal{M}\mathsf{P}$ is stored in, and executed from, ROM.

We now consider two specific types of malware.

Definition 6.2. Self-relocating 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 6.3. 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 $t_s \leq t \leq t_e$, self-relocating malware cannot escape detection. Furthermore, if $t = t_s$, neither can transient malware.

6.4 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 [45] microkernel.

Three points in the timeline of computation of an integrity-ensuring function F are particularly relevant to our discussion (see also Figure 6.4):

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 $\mathcal{P}\mathsf{rv}$ is explicitly requested to *release* an existing lock. This release request might come from $\mathcal{P}\mathsf{rv}$ itself, for instance if R is no longer relevant.

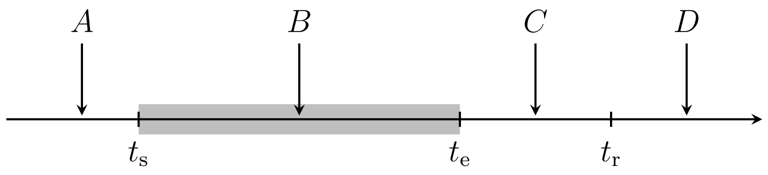


Figure 6.4: Timeline for computation of $R = F(M)$. Computation starts at t_s and ends at t_e. Consistency of R is considered until t_r. A change to M at time A or D has no effect. Impact of a change at time B or C depends on the consistency mechanism.

6.4.1 Simple Approaches

We begin with three obvious options.

`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 self-relocating or transient malware; see Table 6.1.

`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 $[t_\mathrm{s}, t_\mathrm{e}]$. This also implies that M is immutable and thus constant from t_s to t_e.

`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 $[t_\mathrm{s}, t_\mathrm{r}]$, and M stays constant from t_s to t_r. An extended lock can be advantageous if the verifier wishes to guarantee that $\mathcal{P}\mathsf{rv}$ is in a given state at a particular time t_r, as opposed to "some time in the past".

6.4.2 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 6.5.

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.

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 self-relocating, 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.

Table 6.1: Malware detection features.

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

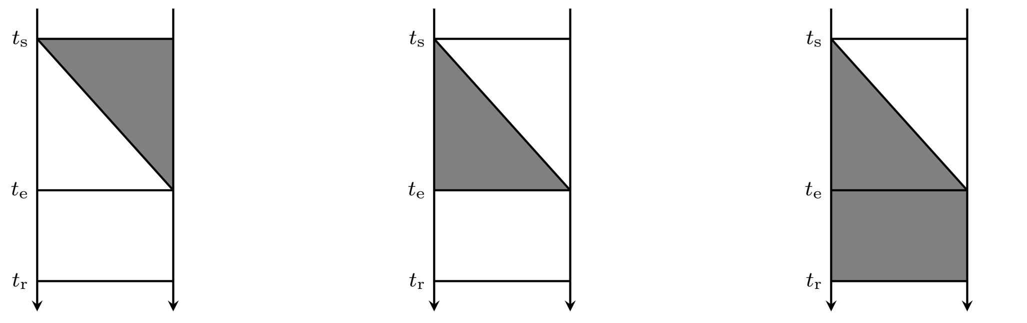
As discussed in Section 6.4.4, `Inc-Lock` is better-suited for handling non-sequential functions. On the other hand, locking M can influence the value of the end-result R. In contrast, `Dec-Lock` guarantees consistency at t_{s} when locking has no impact on R. We consider this to be a subtle yet important 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).

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 $[t_{\mathrm{e}}, t_{\mathrm{r}}]$, and M stays constant in $[t_{\mathrm{e}}, t_{\mathrm{r}}]$. This type of extension is not naturally applicable to `Dec-Lock` since memory is not locked at t_{e}.

6.4.3 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



(a) `Dec-Lock` (b) `Inc-Lock` (c) `Inc-Lock-Ext`

Figure 6.5: Sliding mechanisms discussed in Section 6.4.2. M is represented horizontally. Locked portion of M is in gray.

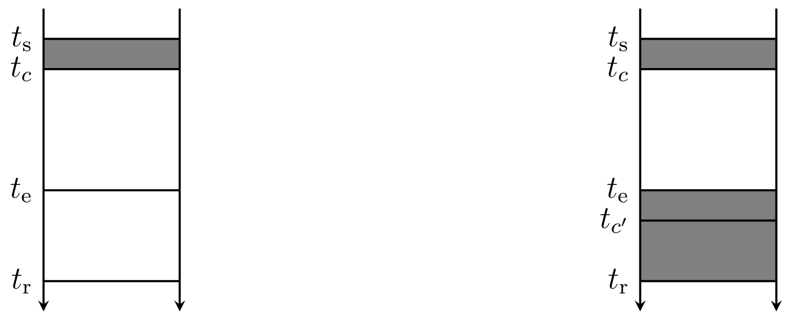
in Figure 6.6.

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 $R = F(M')$.

The same guarantees as `All-Lock` apply here: R is consistent with M in $[t_\mathrm{s}, t_c]$.

`Cpy-Lock` only makes sense if $t_c < t_\mathrm{e}$, 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.



(a) `Cpy-Lock` (b) `Cpy-Lock` & Writeback

Figure 6.6: Mechanisms discussed in Section 6.4.3. M is represented horizontally. Locked portion of M is in gray.

`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`.

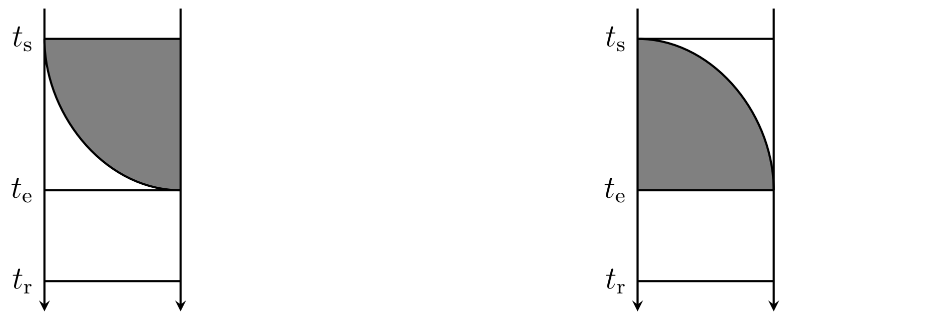
6.4.4 Variations on the Theme

We outline some extensions to previously discussed mechanisms.

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



(a) `Dec-Lock` (Non-Sequential) (b) `Inc-Lock` (Non-Sequential)

Figure 6.7: Locked memory in sliding mechanisms of Section 6.4.2 for non-sequential F. Time goes from top to bottom, and M is represented horizontally. Locked portion of M is in gray.

functions, locks may be acquired sooner, or released later, than in sequential functions. Figure 6.7 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).

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 ($t_\mathrm{s} \leq t_k \leq t_\mathrm{e}$), 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.

Lazy Copy (`Cpy-Lazy`)

Another variation of copy-based mechanisms in Section 6.4.3 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`.

6.4.5 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 [31]) 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.

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

- 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`.

6.4.6 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 6.4.4. 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.

6.4.7 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 6.7.4.

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.

6.5 Temporal Consistency Security Game

In this section, we develop a new definition for a security game that captures temporal consistency in the context of secure remote attestation. We build upon the theoretical model of a processor architecture and syntax from [54]. The work in [54] 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 [54], since we do not require any modification to the underlying processor architecture, as evident in our implementation.

6.5.1 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 $\mathcal{M}\mathsf{P}$.
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 $\mathcal{M}\mathsf{P}$ 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.

6.5.2 Syntax of a Consistent Integrity-Ensuring Measurement Scheme

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

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

6.5.3 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.

Table 6.2: Notation

$\mathcal{A}$	The adversary
$\mathcal{C}$	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`
$\mathcal{M}\mathsf{P}$	Integrity-ensuring measurement function that runs $\mathsf{Respond}$ algorithm.

Recall that, in Section 6.2, we described a typical RA scheme as follows:

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

Definition 6.4. *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 6.8 contains the definition of the security game for a consistent integrity-ensuring measuring scheme ($\mathcal{CMP}$).

Shared by $\mathcal{A}$ and $\mathcal{C}$: random access machine $\texttt{RAM} = (\texttt{M}, \texttt{CPU})$, program W, integrity ensuring measurement function $\mathcal{M}\mathsf{P}$ (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 $\mathcal{C}$:
   - Inputs: $x = x_1 || \ldots || x_i$ for $\texttt{RAM}$.
   - 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. $\mathcal{C}$ runs $\mathsf{Gen}(\lambda)$ to generate setup parameters.
3. $\mathcal{C}$ 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. $\mathcal{C}$ initiates $\mathcal{CMP}$ by generating a challenge c by invoking $\mathsf{Challenge}$ and writing it to the input register. $\mathcal{C}$ invokes `ROM` which contain executable code of $\mathcal{M}\mathsf{P}$. $\mathcal{C}$ 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. $\mathcal{C}$ interprets data in output register as r, a response to its challenge, and outputs bit b, which is the result of $\mathsf{Verify}(c, r, K)$.

Figure 6.8: $\mathcal{CMP}$ Security Game

6.6 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 $\mathcal{M}\mathsf{P}$.

6.6.1 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 ($\mathcal{M}\mathsf{P}$ is secure): A secure HMAC is used to implement $\mathcal{M}\mathsf{P}$.

6.6.2 Proof Sketch for `Dec-Lock`

Considering the security game in Figure 6.8, 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, $\mathcal{M}\mathsf{P}$ computes r on `MEM` and $\mathsf{Verify}(c, r, K)$ should fail, i.e., $b = 0$. b would be 0 because v is now in `MEM` before $\mathcal{M}\mathsf{P}$ starts. Thus, the value of r will indicate that; otherwise, $\mathcal{M}\mathsf{P}$ is insecure, which contradicts Assumption-3. Computation, intermediate and final results of $\mathcal{M}\mathsf{P}$ 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 $t_{\text{vs}} = t_{\text{pr}} = t_{\text{cs}} = \rho_{init}$, and incrementally unlocked as computation of $\mathcal{M}\mathsf{P}$ 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 $\mathcal{M}\mathsf{P}$ 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.

6.6.3 Proof Sketch for `All-Lock`

Considering the security game in Figure 6.8, 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; $\mathsf{Verify}(c, r, K)$ will succeed and produce $b = 1$.

6.7 Implementation & Evaluation

Our prototype of temporal consistency mechanisms is realized in the context of HYDRA hybrid $\mathcal{R}\mathsf{A}$ architecture [30]. The design and implementation of HYDRA are discussed extensively in Chapter 4. Below, we describe implementation details of each mechanism and assess their performance on two popular low- to medium-end development boards: I.MX6-SabreLite [10] and ODROID-XU4 [37].

6.7.1 Experimental Setup

Our implementation ensures temporal consistency by locking memory regions. It thus does not require the execution of the attestation process (PR_{Att}) to be uninterruptible, unlike the original HYDRA implementation [30]. As a result, all user-space processes, including PR_{Att}, have the same priority in our implementation.

The microkernel executable is compiled from the unmodified seL4 source code v4.0.0 [59]. Our user-space code is based on open-source seL4 libraries [58], mostly for providing abstractions for processes, memory management and virtual address space.

6.7.2 Experimental Results: Primitives

Our implementation of mechanisms discussed in Section 6.4 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, PR_{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,

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

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 [75], AES256-CBC based MAC [41] and HMAC-SHA256 [61] algorithms.

Figure 6.9 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.

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 6.10 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.

6.7.3 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.

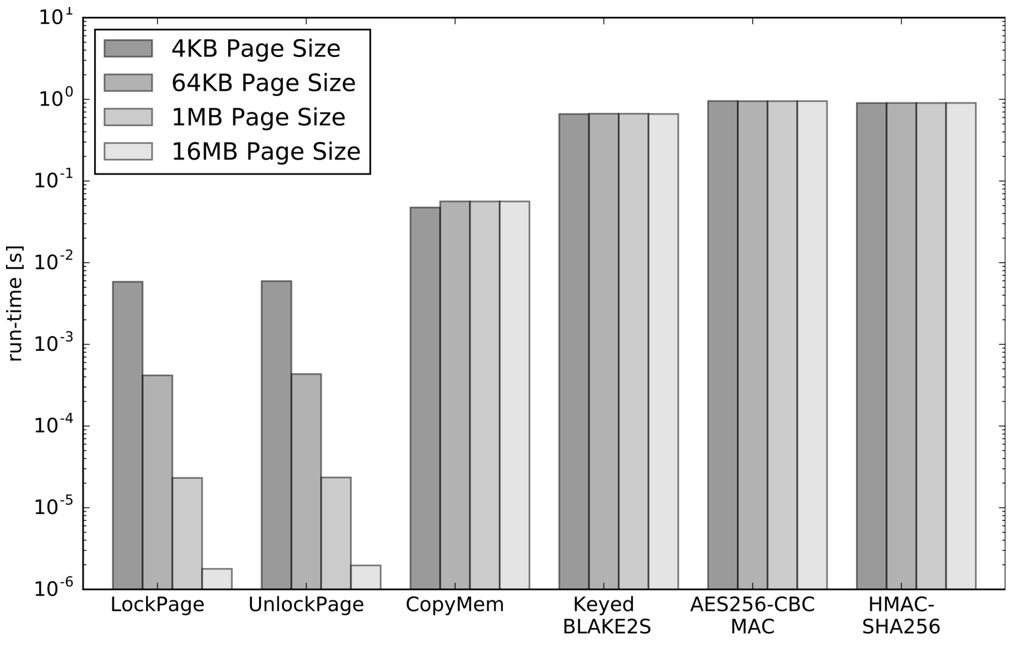


Figure 6.9: Performance of primitives with 16MB of memory on I.MX6-SabreLite.

Figure 6.11 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 PR_{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

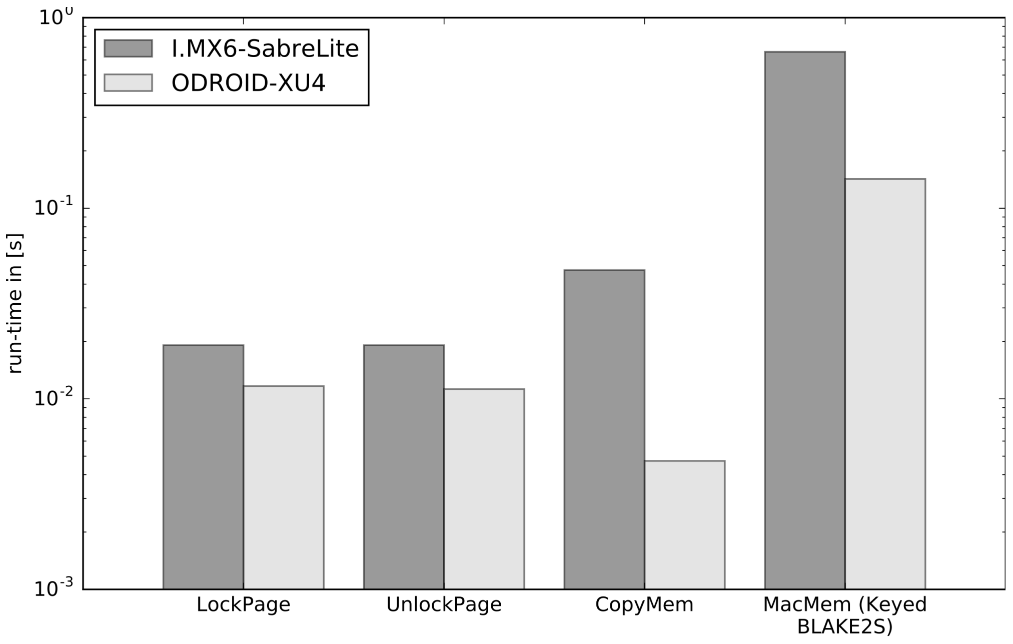
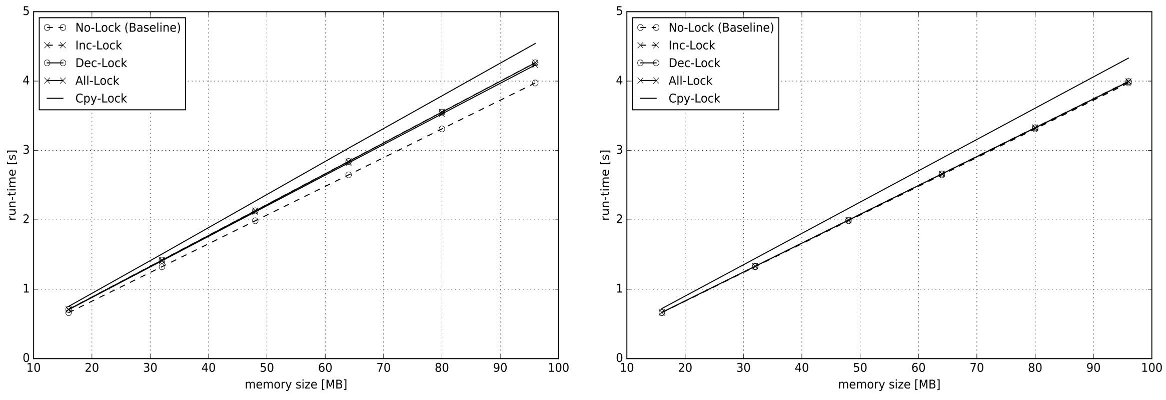


Figure 6.10: Performance of primitives with 16MB memory on I.MX6-SabreLite and ODROID-XU4.

large page sizes. One disadvantage of larger page sizes is that memory pages, on average, will be locked for longer periods.

- `Cpy-Lock` comes out as the preferred mechanism. It incurs small ($\sim 8\%$) 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'.



(a) 4KB Page Size (b) 64KB Page Size

Figure 6.11: Run-time of various temporal consistency ensuring mechanisms in I.MX6-SabreLite.

6.7.4 Implementation of Inconsistency Detection

We could implement the inconsistency detection mechanism by having PR_{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 PR_{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, PR_{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 PR_{Att} will be made aware of such inconsistency; PR_{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 6.7.5 that the overhead from this interference is very small compared to the actual measurement runtime.

To implement this mechanism in HYDRA, we decompose PR_{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 $\mathcal{V}\mathsf{rf}$ when an inconsistency occurs.

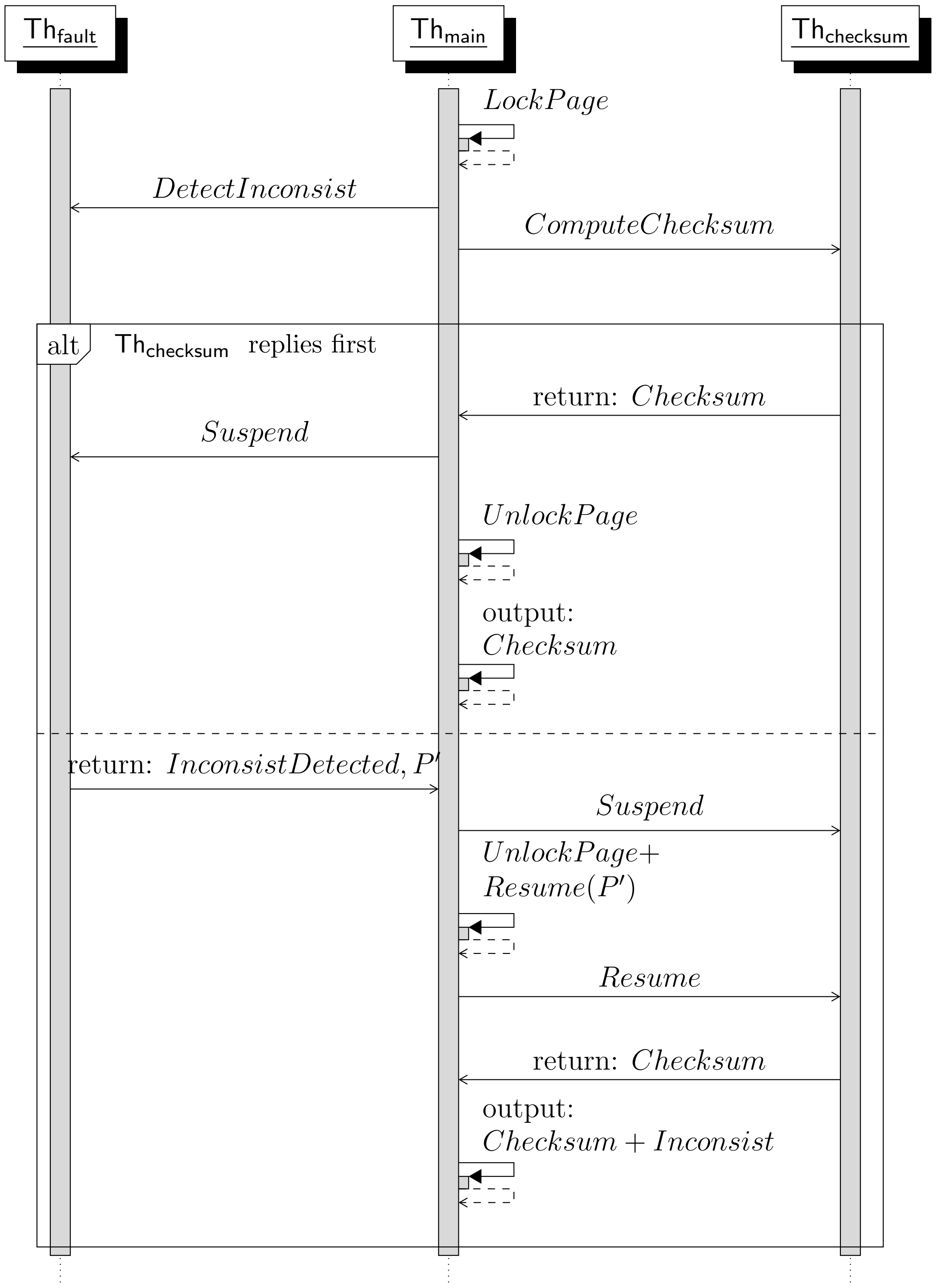

$\mathsf{Th}_{\mathsf{fault}}$
$\mathsf{Th}_{\mathsf{main}}$
$\mathsf{Th}_{\mathsf{checksum}}$
$LockPage$
$DetectInconsist$
$ComputeChecksum$
alt
$\mathsf{Th}_{\mathsf{checksum}}$ replies first
return: $Checksum$
$Suspend$
$UnlockPage$
output: $Checksum$
return: $InconsistDetected, P'$
$Suspend$
$UnlockPage+$ $Resume(P')$
$Resume$
return: $Checksum$
output: $Checksum + Inconsist$

Figure 6.12: Sequence diagram of PR_{Att} with memory inconsistency detection during single attestation instance. PR_{Att} chooses to resume execution of $\mathsf{Th}_{\mathsf{checksum}}$ after P' causes memory inconsistency.

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 message indicates the attempted write access on memory being attested. A sample code for $\mathsf{Th}_{\mathsf{fault}}$ is provided in Listing 6.1.

```
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))
```

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

```
    {
      /* 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);
}
```

Listing 6.1: Sample code for $\mathsf{Th}_{\mathsf{fault}}$

A diagram in Figure 6.12 summarizes the sequence of operation of our modified PR_{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 $\mathcal{V}\mathsf{rf}$.

6.7.5 Experimental Results: Inconsistency Detection

To evaluate performance of the inconsistency detection mechanism, we experimented by running two processes – modified PR_{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 PR_{Att} running at any given time.

Results in Figure 6.13 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 PR_{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

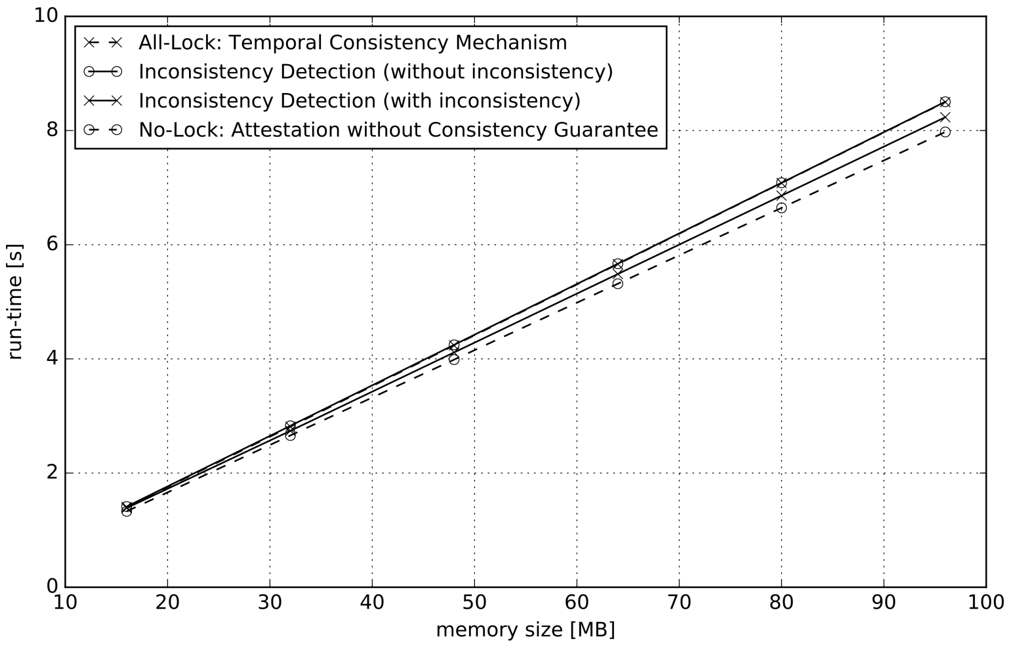


Figure 6.13: Run-time of inconsistency detection with 4KB page size on I.MX6-SabreLite.

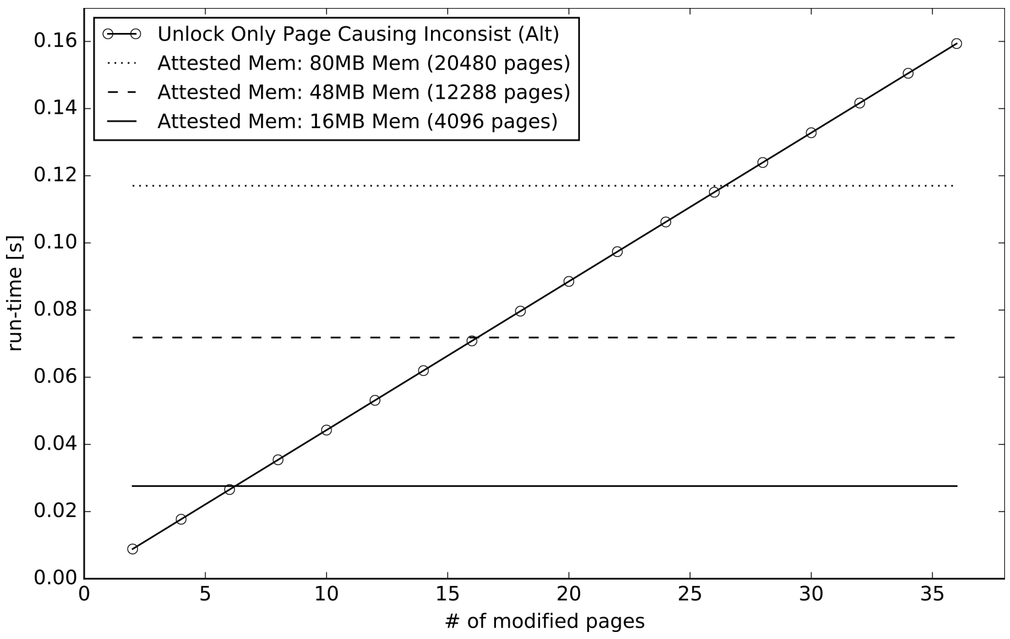


Figure 6.14: Downtime of the faulting process P' when its actions result in an inconsistency with 4KB page size on I.MX6-SabreLite. Horizontal lines represent downtime from the approach where PR_{Att} resolves inconsistency by unlocking entire memory of P'.

P' is being suspended. This results in run-time of the unlock operation being $\sim$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 PR_{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 6.14 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.

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

Chapter 7

Periodic Self-Measurement

7.1 Introduction

In recent years, embedded and cyber-physical systems (CPS), under the guise of *Internet-of-Things (IoT)*, have entered many aspects of daily life, including: homes, office buildings, public venues, factories and vehicles. This trend of adding computerized components to previously analog devices and then inter-connecting them brings many obvious benefits. However, it also greatly expands so-called *"attack surface"* and turns these newly computerized gadgets into natural and attractive attack targets. In particular, as the 2016 Mirai botnet demonstrated, IoT devices can be infected with malware and used as bot-controlled *zombies* in Distributed Denial-of-Service (DDoS) attacks [4]. Also, IoT-borne malware can snoop on device owners (by sensing) or maliciously control critical services (by actuation), as happened with Stuxnet [87].

One key component in securing IoT devices is malware detection, which is typically attained with remote attestation. Remote attestation is a distinct security service that allows a trusted party, called *verifier* ($\mathcal{V}\mathsf{rf}$), to securely verify the internal state (including memory and storage) of a remote untrusted and potentially malware-infected device, called *prover* ($\mathcal{P}\mathsf{rv}$). $\mathcal{R}\mathsf{A}$ is generally realized as an interactive protocol between $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$. A typical example is described in [12]: (1) $\mathcal{V}\mathsf{rf}$ sends an attestation request to $\mathcal{P}\mathsf{rv}$, (2) $\mathcal{P}\mathsf{rv}$ verifies the request[1], (3) computes a cryptographic function of its internal state, (4) sends the result to $\mathcal{V}\mathsf{rf}$, and (5) $\mathcal{V}\mathsf{rf}$ finally checks the result and decides whether $\mathcal{P}\mathsf{rv}$ is infected.

This general approach is referred to as *on-demand attestation* and all current $\mathcal{R}\mathsf{A}$ techniques adhere to it. In this chapter, we identify two important limitations of such approach. First, it is a poor match for unattended devices, since malware that "comes and goes" (i.e., mobile malware [68]) can not be detected if it leaves $\mathcal{P}\mathsf{rv}$ by the time attestation is performed. Second, for a device working under time constraints (real-time operation) or otherwise providing safety-critical services, on-demand attestation requires performing a possibly time-consuming task while deviating from the device's main function(s).

To address these issues, we design ERASMUS: **E**fficient **R**emote **A**ttestation via **S**elf-**M**easurement for **U**nattended **S**ettings. ERASMUS is based on *periodic self-measurements.* In it, $\mathcal{P}\mathsf{rv}$ measures and records its state at scheduled times. Measurements are stored in $\mathcal{P}\mathsf{rv}$'s *insecure* memory. $\mathcal{V}\mathsf{rf}$ occasionally collects and validates these measurements in order to establish the *history* of $\mathcal{P}\mathsf{rv}$'s state. In this general approach, $\mathcal{V}\mathsf{rf}$ imposes only negligible real-time burden on $\mathcal{P}\mathsf{rv}$. It also offers strictly better quality-of-service than prior attestation techniques, because $\mathcal{V}\mathsf{rf}$ obtains $\mathcal{P}\mathsf{rv}$'s entire history of measurements, since the last $\mathcal{V}\mathsf{rf}$ request. In other words, ERASMUS de-couples (1) frequency of $\mathcal{P}\mathsf{rv}$ checking, from (2) frequency of $\mathcal{P}\mathsf{rv}$ measurements, which are equivalent in on-demand attestation. Finally, ERASMUS simplifies $\mathcal{R}\mathsf{A}$ design (in terms of required features) for $\mathcal{P}\mathsf{rv}$: authentication of $\mathcal{V}\mathsf{rf}$ requests is no longer needed, since computational DoS attacks do not arise. This differs from requirements in [12] that stipulate (potentially expensive) $\mathcal{P}\mathsf{rv}$ authentication of all $\mathcal{V}\mathsf{rf}$'s requests.

[1]Since attestation is a potentially expensive task, this relatively light-weight verification mitigates computational DoS attacks.

We also introduce the new notion of *Quality of Attestation* (QoA) which captures: (1) how $\mathcal{P}\mathsf{rv}$ is attested, (2) how often its state is measured, and (3) how often these measurements are verified. It is the temporal analogue of the concept of quality of swarm attestation (QoSA) introduced in [16] in the context of attesting groups of devices.

NOTE: ERASMUS is not intended as a replacement for on-demand attestation. Clearly, for some devices, and in some settings, real-time on-demand attestation is mandatory, e.g., immediately before or after a software update, or in the context of secure erasure/reset. Also, on-demand attestation may be more flexible, e.g., if $\mathcal{V}\mathsf{rf}$ is only interested in measuring a fraction of $\mathcal{P}\mathsf{rv}$'s memory. These two approaches are not mutually exclusive and may be used together to increase QoA, specifically, in terms of freshness of the latest measurement.

The last incentive for the self-measurement approach is its suitability for highly mobile groups of devices. $\mathcal{R}\mathsf{A}$ protocols developed for "swarm attestation", e.g., [7, 76, 38, 16], are designed to efficiently attest groups of interconnected devices on-demand, with a single interaction between $\mathcal{V}\mathsf{rf}$ and multiple $\mathcal{P}\mathsf{rv}$-s. However, such protocols do not work in highly mobile swarms, since on-demand attestation requires topology to essentially remain static during the entire attestation protocol instance, duration of which is dominated by computation on all swarm devices. Since ERASMUS involves virtually no real-time computation for $\mathcal{P}\mathsf{rv}$, it is more suitable for high-mobility swarm settings.

7.2 Remote Attestation via Self-Measurements

As discussed in the previous section, all current techniques perform *on-demand* $\mathcal{R}\mathsf{A}$, This can be a time-consuming activity that diverts $\mathcal{P}\mathsf{rv}$'s attention away from its primary mission. However, $\mathcal{P}\mathsf{rv}$ performs no $\mathcal{R}\mathsf{A}$ computation between successive $\mathcal{V}\mathsf{rf}$'s requests. In contrast, ERASMUS divides $\mathcal{R}\mathsf{A}$ into two phases. In the *measurement* phase, $\mathcal{P}\mathsf{rv}$ performs self-measurements based on a pre-established schedule, and stores the results. In the collection phase, $\mathcal{V}$rf (whenever it chooses to do so) contacts $\mathcal{P}$rv to fetch these measurements. The collection phase is very fast since it requires practically no computation by $\mathcal{P}$rv. In particular, since measurements are based on a MAC computed with a key shared between $\mathcal{P}$rv and $\mathcal{V}$rf, no extra protection is needed when $\mathcal{P}$rv sends measurements to $\mathcal{V}$rf. Furthermore, unlike on-demand attestation, there is no threat of computational DoS on $\mathcal{P}$rv, and therefore no need to authenticate $\mathcal{V}$rf's requests.

A $\mathcal{P}$rv's measurement M_t computed at time t is defined as:

$$M_t =< t, H(mem_t), \mathrm{MAC}_K(t, H(mem_t)) >$$

where H is a suitable cryptographic hash (e.g., SHA-256) function and mem_t represents $\mathcal{P}$rv's memory at time t. Computation of $H(mem_t)$ and MAC is done in the context of the underlying $\mathcal{R}$A architecture, e.g., SMART+ or HYDRA.

Although ERASMUS assumes a symmetric key K shared between $\mathcal{V}$rf and $\mathcal{P}$rv, a public key signature scheme could be used instead, with no real impact on security of the scheme except for the higher cost of measurements.

7.2.1 Quality of Attestation

Quality of Attestation (QoA) is primarily determined by two parameters: (1) time T_M between two successive *measurements* on $\mathcal{P}$rv, and (2) time T_C between two successive requests by $\mathcal{V}$rf to collect measurements from $\mathcal{P}$rv.

We assume that, in most cases, $T_C > T_M$. If happens that $T_C \leq T_M$, $\mathcal{V}$rf simply collects the same measurements more than once, which is redundant. Alternatively, $\mathcal{V}$rf can **explicitly request** $\mathcal{P}$rv to perform a fresh measurement before the collection. In that case, $\mathcal{V}$rf's request

Table 7.1: Quality of Attestation Parameters

Notation	Meaning	Trade-off between:
T_M	Time between two consecutive measurements	+ High detection rate - High burden on $\mathcal{P}\mathsf{rv}$
T_C	Time between two consecutive $\mathcal{V}\mathsf{rf}$ requests	+ Fast detection by $\mathcal{V}\mathsf{rf}$ - High burden on $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$
f	Freshness of the latest measurement	+ Fresh measurement - High burden on $\mathcal{P}\mathsf{rv}$

would have to be authenticated and checked for freshness (as in SMART+ [12]) before the on-demand measurement is computed. These activities clearly incur additional real-time overhead and delays. This variant is called ERASMUS+OD and it is discussed in Section 7.3.

Exactly how T_C and T_M are determined depends on the specifics of $\mathcal{P}\mathsf{rv}$'s mission and its deployment setting. Security impact of these parameters is intuitive. Smaller T_M implies smaller window of opportunity for mobile malware to escape detection. Smaller T_C implies faster malware detection. If either value is large, attestation becomes ineffective. Meanwhile, though low values increase QoA, they also increase $\mathcal{P}\mathsf{rv}$'s overall burden, in terms of computation, power consumption and communication.

Without loss of generality, we assume that measurements and collections occur at regular intervals. Of course, in practice this might not work for critical or time-sensitive applications (see Section 7.5). In fact, it might be advantageous to take measurements at irregular intervals, since doing so might give $\mathcal{P}\mathsf{rv}$ a bit of an extra edge against mobile malware, as discussed in Section 7.2.4.

Another ERASMUS parameter is the number of measurements (referred to as k) obtained by $\mathcal{V}\mathsf{rf}$ in each collection phase. It can range between *one* (only the most recent measurement) and *all*. In a typical setting, $\mathcal{P}\mathsf{rv}$'s history size should be set such that each measurement is

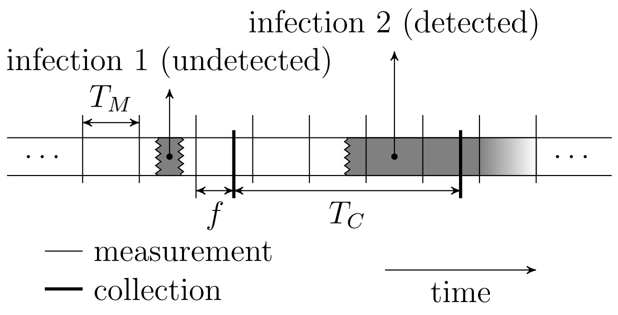


Figure 7.1: QoA illustration: Infection 1 by mobile malware is undetected; Infection 2 is detected. T_M is the time between two measurements, T_C is the time between two collections, and f is the freshness of each measurement.

collected exactly once. That is, $k = \lceil T_C/T_M \rceil$.

Finally, the collection phase involves the notion of *freshness*, i.e., how recent is $\mathcal{P}\mathsf{rv}$'s latest measurement. Depending on the application, maximal freshness might be required, e.g., right before or after a software update. Maximal freshness is attainable via on-demand attestation. In ERASMUS, freshness of a measurement (denoted as f) ranges between T_M and 0, which correspond to minimal and maximal freshness, respectively. On average, we expect $f = T_M/2$.

Figure 7.1 shows an example with two malware infections. In the first, malware covers its tracks and leaves before any measurement takes place. In the second, malware persists on $\mathcal{P}\mathsf{rv}$. Although a measurement occurs perhaps soon after infection, corrective action can be taken only after collection, thus illustrating the importance of a small T_C. Measurements and collections are shown as punctual events in Figure 7.1. Although they do take some time to complete (measurements, in particular), they are considered negligible in $\mathcal{P}\mathsf{rv}$'s overall lifecycle (see Section 7.4). We summarize QoA parameters and their security implications in Table 7.1.

7.2.2 Measurements Storage & Collection

A naïve way for $\mathcal{P}\mathsf{rv}$ to store measurements is to keep track of them indefinitely. However, this will eventually consume a lot of $\mathcal{P}\mathsf{rv}$'s storage. To this end, ERASMUS uses *rolling measurements*. A fixed section of $\mathcal{P}\mathsf{rv}$'s insecure storage is allocated as a windowed (circular) buffer for n measurements. The i-th measurement is stored at location $L_{i \bmod n}$. However, it is expected that $\mathcal{V}\mathsf{rf}$ collects measurements sufficiently often, such that no measurement is over-written. That is, the time between successive collections should be at most $T_C \leq n \cdot T_M$.

The interaction between $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$ is very simple: $\mathcal{V}\mathsf{rf}$ asks for the k latest measurements, which $\mathcal{P}\mathsf{rv}$ simply reads from the buffer and transmits. The collection phase does not involve any change of state on $\mathcal{P}\mathsf{rv}$ and returned measurements are not encrypted. (However, recall that they are authenticated since each measurement is a MAC computed using K). It also does not trigger any significant computation on $\mathcal{P}\mathsf{rv}$, i.e., in contrast with on-demand attestation, no cryptographic operations are required in the collection phase.

Self-measurements can be stored in $\mathcal{P}\mathsf{rv}$'s unprotected storage. This allows malware (that is possibly present on $\mathcal{P}\mathsf{rv}$) to tamper with measurements, by modifying, re-ordering and/or deleting them. However, since malware (by design of SMART) cannot access K, it cannot forge measurements. Thus, it is easy to see that any tampering will be detected by $\mathcal{V}\mathsf{rf}$ at the next collection phase and hence malware presence would be noticed. For same reasons, code that handles request parsing as well as storage and transmission of measurements does not need to be executed in a secure environment, or stored in ROM. Code that performs self-measurement, however, must be protected by the underlying security architecture, as in on-demand attestation.

Scheduling in ERASMUS can be implemented in a very simple and stateless manner. Let t be the time specified by a Reliable Read-Only Clock (RROC) at measurement M_t, and let T_M be the time between two successive measurements, as configured in $\mathcal{P}\mathsf{rv}$. The windowed

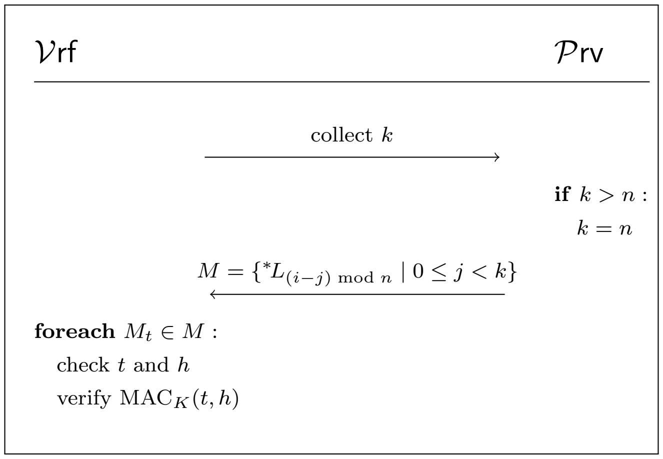

$\mathcal{V}$rf
$\mathcal{P}$rv
collect k
if $k > n$:
$k = n$
$M = \{{}^*L_{(i-j) \bmod n} \mid 0 \leq j < k\}$
foreach $M_t \in M$:
check t and h
verify $\text{MAC}_K(t, h)$

Figure 7.2: ERASMUS collection protocol.

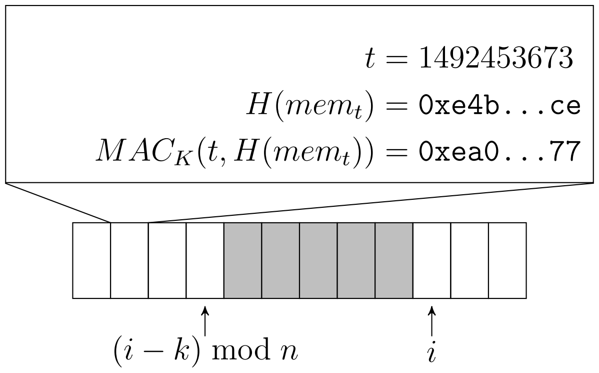

$t = 1492453673$
$H(mem_t) = \texttt{0xe4b...ce}$
$MAC_K(t, H(mem_t)) = \texttt{0xea0...77}$
$(i-k) \bmod n$
i

Figure 7.3: ERASMUS memory allocation. Example with $n = 12, i = 3, k = 7$.

buffer slot L_i, used to store M_t, is determined by: $i = \lfloor t/T_M \rfloor \bmod n$.

ERASMUS collection protocol is shown in Figure 7.2. The underlying attestation architecture is not involved in the collection phase. Notation ${}^*L_j$ refers to contents of memory location L_j. A sample memory layout is shown in Figure 7.3.

7.2.3 Security Considerations

Security of the measurement process itself is based on the underlying security architecture, e.g., SMART+ or HYDRA, which: (1) provides measurement code with exclusive access to

K, (2) ensures non-malleability and non-interruptibility of the measurement code, and (3) performs memory-cleanup after execution.

Timestamps used in the measurement process *must* be obtained from a Reliable Read-Only Clock (RROC), which (by definition) can not be modified by non-physical means. This is important, since malware should not influence *when* measurements are taken. If RROC values could be modified, the following attack scenario would become possible: malware enters at time t_0 and remains activeuntil a measurement at time $t_0 + \delta$ (with $\delta < T_M$) is taken. Before leaving, malware discards that measurement and resets the counter to t_0. Soon after δ (so that a measurement, valid this time, has been taken for $t_0 + \delta$), malware returns and resets the counter to time elapsed since t_0. Though this example works for one T_M window, it can be extended to arbitrarily many. It requires an additional assumption that no collection took place during the presence of malware.

Fortunately, RROC is already a requirement for SMART+, for a totally different reason: RROC helps prevent replay and computational DoS attacks on $\mathcal{P}$rv. Thus, ERASMUS does not require any changes to the underlying security architecture.

As mentioned earlier, measurements need not be stored in protected memory because tampering is detectable and indicates malware presence on $\mathcal{P}$rv. Likewise, the code to support the collection phase does not require any protection since measurements are not secret (they are unique for every device and every timestamp value), and their absence or alteration is self-incriminating.

7.2.4 Irregular Intervals

A natural extension to ERASMUS is the use of irregular measurement intervals, instead of a fixed T_M. The main motivation is that mobile malware that is aware of fixed scheduling

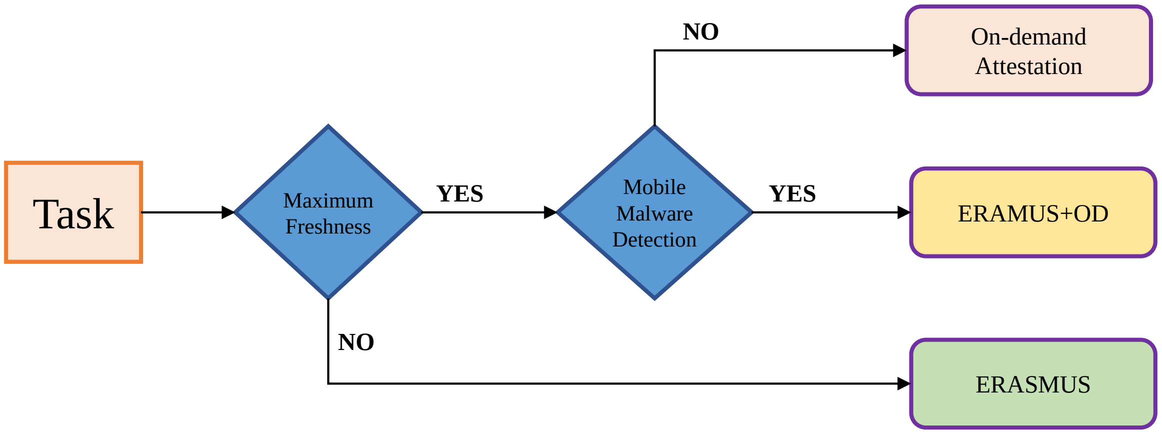


Figure 7.4: Diagram summarizing major $\mathcal{RA}$ tasks of ERASMUS and on-demand attestation

knows when to enter/leave $\mathcal{P}$rv in order to stay undetected. One way to implement irregular intervals is via Cryptographic Pseudo Random Number Generator (CPRNG) [53] initialized (seeded) with the secret key K. Output of the CPRNG can be truncated, such that T_M is upper- and/or lower-bounded.

For example, after computing M_{t_i}, $\mathcal{P}$rv can set the measurement timer to:

$$T_M^{\text{next}} = \text{map}(\text{CPRNG}_k(t_i)),$$

where map is a function that maps CPRNG output to seconds, e.g., map : $x \mapsto x \bmod (U - L) + L$, with U and L upper and lower bounds, respectively. The timer itself must be read-protected to ensure that T_M^{next} is unknown to malware potentially present on $\mathcal{P}$rv. CPRNG code must be protected the same way as the measurement code.

7.3 Comparison with On-demand Attestation

We now discuss advantages and disadvantages of ERASMUS as compared to on-demand attestation. We also propose a method to combine both techniques while retaining most of their respective benefits. Figure 7.4 summarizes the discussion throughout this section.

ERASMUS has several advantages over on-demand attestation. Most importantly, it enables detection of mobile malware on $\mathcal{P}\mathsf{rv}$. Measurements can be performed more often without increased $\mathcal{V}\mathsf{rf}$ participation. This is an important security consideration, since frequency of (self-)measurements determines the window of opportunity for mobile malware. Another advantage comes from the fact that the collection phase of ERASMUS requires practically no computation on $\mathcal{P}\mathsf{rv}$. This makes ERASMUS inherently resilient against computational DoS attacks, without explicitly authenticating $\mathcal{V}\mathsf{rf}$'s requests. Finally, measurement scheduling in ERASMUS can be made context-aware. This makes ERASMUS better suited for safety-critical applications; this is discussed further in Section 7.5.

Despite these benefits, ERASMUS does not fully obviate the need for on-demand attestation. In applications where a quick reaction to infection is crucial (e.g., immediately before or after a software update or for secure erasure/reset) on-demand attestation is still necessary, as it is the only way to maximize freshness of attestation, given that verification is performed immediately after the measurement.

Fortunately, ERASMUS may be combined with on-demand attestation to retain advantages of both approaches. This variant, ERASMUS+OD, records $\mathcal{P}\mathsf{rv}$'s state history to detect mobile malware, and uses on-demand attestation to obtain better freshness. Freshness is particularly relevant whenever real-time attestation is mandatory.

The measurement phase is unmodified, while the collection phase is combined with on-demand attestation request as follows: First, as part of each attestation request, $\mathcal{V}\mathsf{rf}$ computes and includes an authentication token and specifies k. As in SMART+ [12], authentication of $\mathcal{V}\mathsf{rf}$ protects $\mathcal{P}\mathsf{rv}$ against computational DoS. Then, only after checking that a request is valid, $\mathcal{P}\mathsf{rv}$ computes a new measurement. Finally, this real-time measurement is sent to $\mathcal{V}\mathsf{rf}$, along with k previous measurements. This protocol is shown in Figure 7.5.

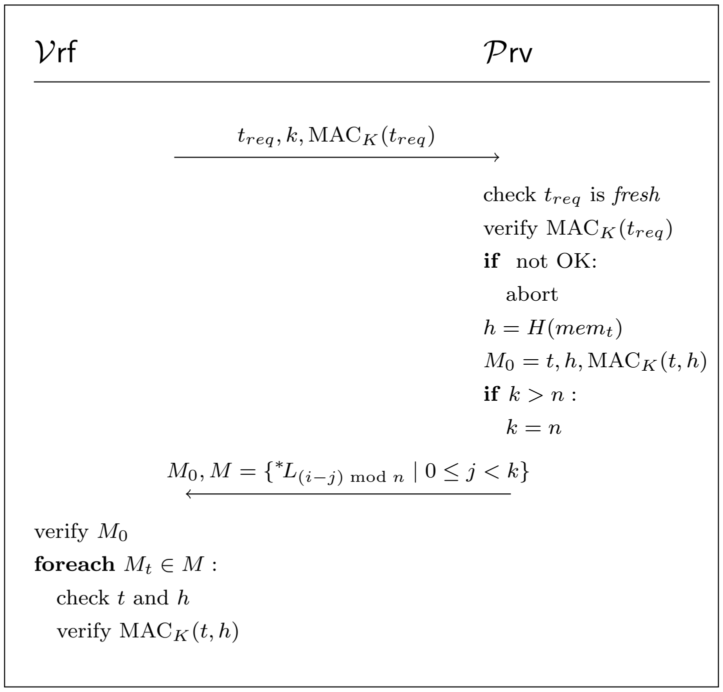

$\mathcal{V}$rf
$\mathcal{P}$rv
$t_{req}, k, \text{MAC}_K(t_{req})$
check t_{req} is *fresh*
verify $\text{MAC}_K(t_{req})$
if not OK:
abort
$h = H(mem_t)$
$M_0 = t, h, \text{MAC}_K(t, h)$
if $k > n$:
$k = n$
$M_0, M = \{{}^*L_{(i-j) \bmod n} \mid 0 \leq j < k\}$
verify M_0
foreach $M_t \in M$:
check t and h
verify $\text{MAC}_K(t, h)$

Figure 7.5: ERASMUS+OD protocol.

7.4 Implementation

We implemented ERASMUS on two security architectures: SMART+ and HYDRA. The main difference is that the former targets low-end devices, and the latter – medium-end devices with a memory management unit (MMU).

7.4.1 Implementation on SMART+

Figure 7.6 shows the implementation of ERASMUS atop SMART+ architecture. As in SMART+, measurement code and K reside in ROM. However, the code is invoked periodically and autonomously, whenever a scheduled timer interrupt occurs. We now examine ROM size, hardware costs and run-time.

ROM Size: This greatly depends on the choice of the MAC algorithm. We implement ROM-

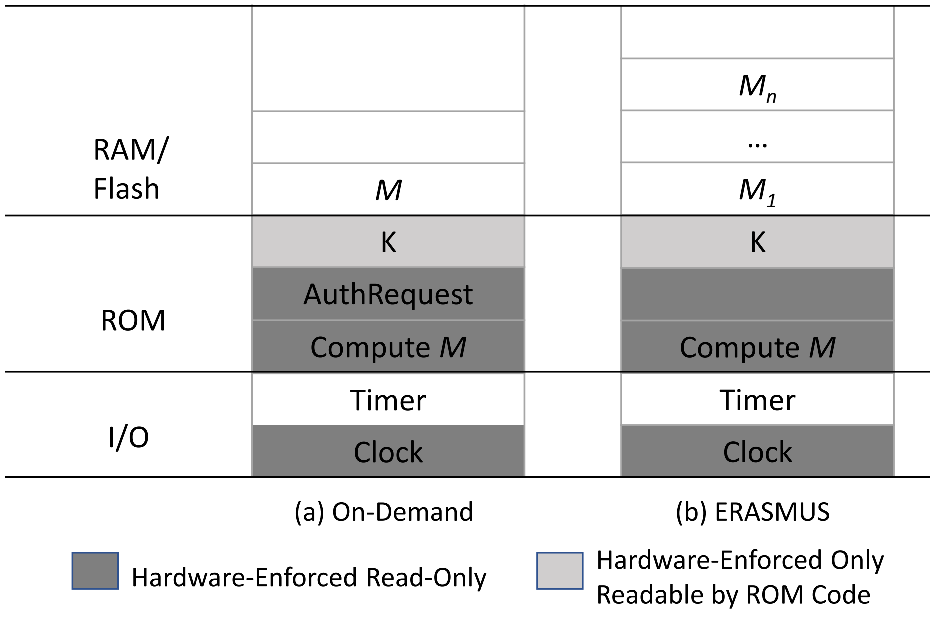


Figure 7.6: Memory organization and access rules of SMART+-based $\mathcal{R}$A. r denotes exclusive read-only access.

Table 7.2: Size of Attestation Executable

MAC Impl.	SMART+		HYDRA	
	On-Demand	ERASMUS	On-Demand	ERASMUS
HMAC-SHA1	4.9KB	4.7KB	-	-
HMAC-SHA256	5.1KB	4.9KB	231.96KB	233.84KB
Keyed BLAKE2S	28.9KB	28.7KB	239.29KB	241.17KB

Table 7.3: Hardware Cost on MSP430

H/W	Original	On-demand	ERASMUS
Register	579	655	655
Look-up Table	1,731	1,969	1,969

resident code in "C" using three MAC functions: HMAC-SHA1 [28][2], HMAC-SHA256 [61] and keyed BLAKE2S [75]. We then use open-source MSP430-gcc compiler [84] to compile the "C" code into an MSP430 executable. Table 7.2 shows the ROM size for each SMART+-based approach. As expected, ERASMUS requires slightly less ROM than on-demand attestation.

[2]Note that HMAC-SHA1 is only used for comparison purposes. We exclude it in our actual implementations due to a recent collision attack on SHA1 [83].

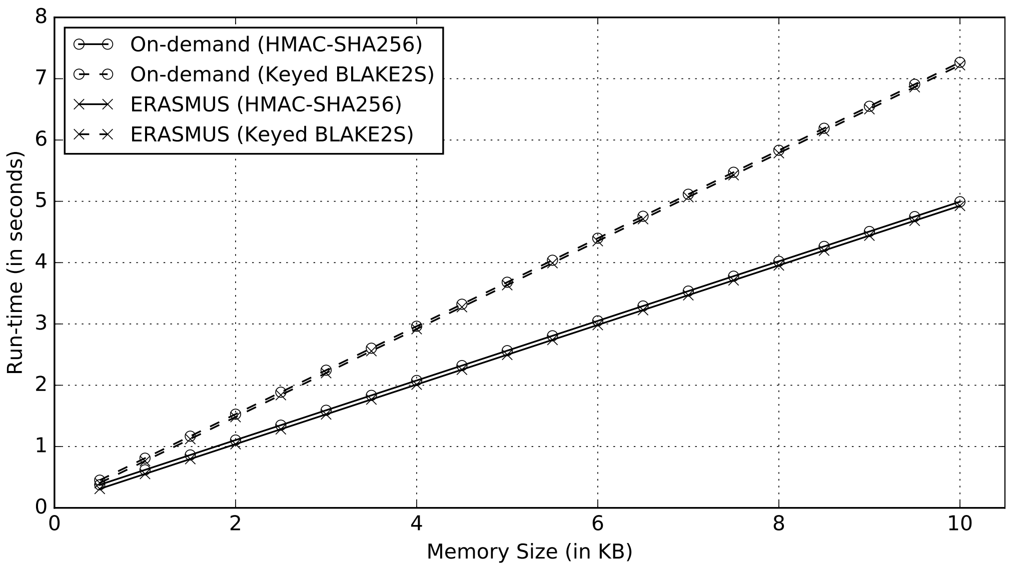


Figure 7.7: Measurement Run-Time on MSP430-based Device @ 8MHz

Hardware Cost: We implement the hardware part of ERASMUS by modifying the MSP430 architecture, using the open-source OpenMSP430 core [35]. We modify the memory backbone module in the OpenMSP430 core to support atomic execution of ROM code and exclusive access to K. RROC is realized as a peripheral using a 64-bit register incremented for every clock cycle. To ensure write-protection, a write-enable wire is removed from the RROC module. For timer components, we use the unmodified version of the `omsp_timerA` module provided by OpenMSP430. Note that we do not consider hardware timers as additional hardware cost because they are common and crucial components of most embedded systems. Indeed, it is unusual to find an embedded device not equipped with at least one timer. Finally, we use Xilinx ISE 14.7 [89] to synthesize our modifications of the OpenMSP430 core from a hardware description language to a combination of registers and look-up tables in an FPGA. As expected, synthesized results in Table 7.3 show that ERASMUS utilizes the same number of registers and look-up tables as on-demand attestation. Compared to the unmodified OpenMSP430 core, ERASMUS requires roughly 13% and 14% additional registers and look-up tables respectively.

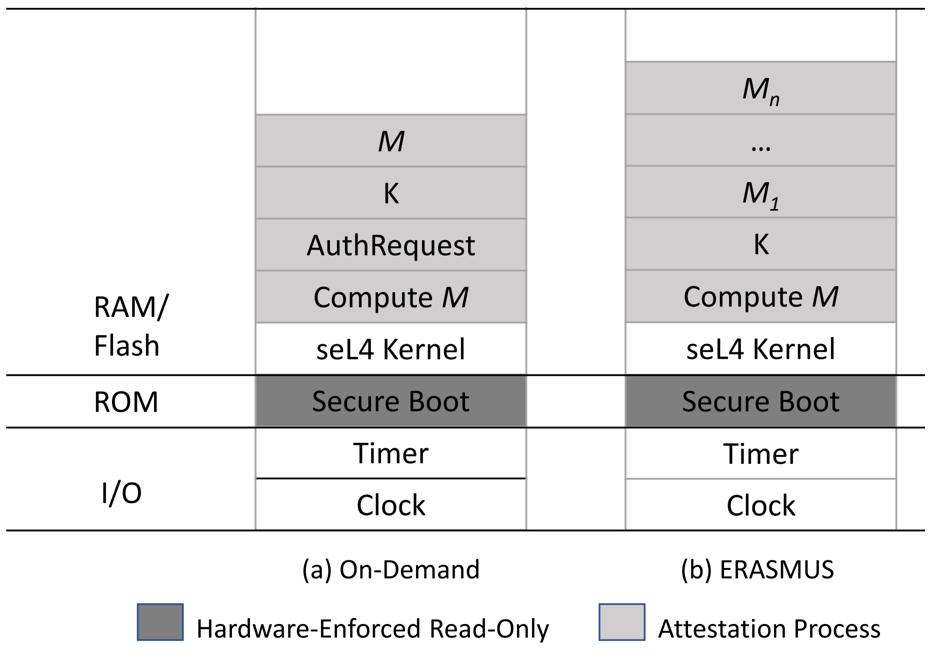


Figure 7.8: Memory organization of HYDRA-based on-demand attestation and ERASMUS.

Measurement Run-Time: Figure 7.7 illustrates run-time of the measurement phase for various memory sizes. Not surprisingly, it is linearly dependent on memory size and roughly equivalent to that of on-demand attestation.

7.4.2 Implementation on HYDRA

Table 7.4: Run-Time (in ms) of Collection Phase on I.MX6-Sabre Lite

Operations	ERASMUS	ERASMUS+OD
Verify Request	N/A	0.005
Compute Measurement[3]	N/A	285.6
Construct UDP Packet	0.003	0.003
Send UDP Packet	0.012	0.012
Total Collection Run-time	0.015	285.6

Figure 7.8 shows implementations of HYDRA-based ERASMUS and on-demand attestation. Both are realized on an I.MX6 Sabre Lite [10] development board. RROC is implemented

[3]On 10MB memory using keyed BLAKE2S as the underlying MAC function.

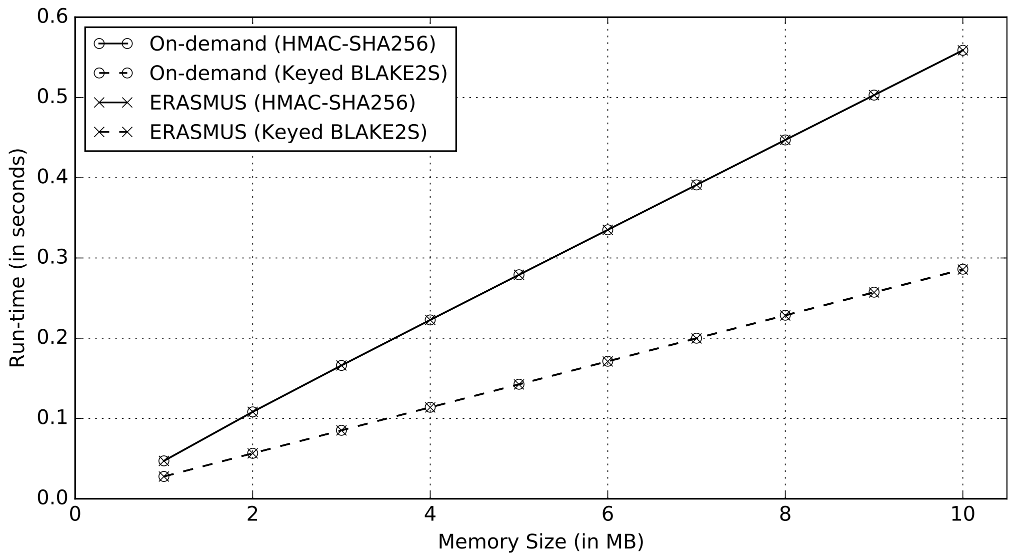


Figure 7.9: Measurement Run-Time on I.MX6 Sabre Lite @ 1GHz

based on the software clock approach, suggested by Brasser et al. [12]. Specifically, we use a short-term counter from Sabre Lite's General Purpose Timer (GPT) and our clock code in HYDRA's attestation process (PR_{Att}) to construct the RROC. When the counter wraps around and causes an interrupt, our clock code handles it by updating higher-order bits of the clock in PR_{Att}. Then, the clock value is constructed by combining these bits with the GPT counter. To ensure the read-only property, PR_{Att} is given exclusive write-access to RROC components. Also, we use Sabre Lite's Enhanced Periodic Interrupt Timer (EPIT) to schedule execution of ERASMUS measurement code

We base the code of PR_{Att} on open-source seL4 libraries [58]: `seL4utils, seL4vka, seL4vspace`, and `seL4bench`. The first three provide abstractions of: process, memory management and virtual space, respectively, while the last one is used to evaluate performance. Finally, we use the code in [60] to implement the network stack, an Ethernet driver and timer drivers in seL4.

Executable Size: Table 7.2 compares executable sizes of PR_{Att} in on-demand attestation and ERASMUS. Results show that ERASMUS is only about 1% larger in terms of executable size. This overhead mostly comes from the need for an additional timer driver.

Measurement Run-time: Measurement run-time of HYDRA-based ERASMUS in Figure 7.9 follows the same trend as SMART+-based ERASMUS: (1) it is linear as a function of memory sizes, and (2) it is roughly equal to that of on-demand attestation. The same figure shows that performing self-measurement (based on keyed BLAKE2S) takes less than 300msec on a 1GHz device with 10MB memory.

Collection Run-time: Table 7.4 shows the run-time breakdown of the collection phase for each variant. Clearly, run-time of the collection phase in ERASMUS is negligible (by at least a factor of $3,000$), compared to that of the measurement phase. Collection run-time in ERASMUS+OD, on the other hand, is dominated by run-time of performing on-demand attestation.

7.5 Availability in Time-Sensitive Applications

In some cases, it might be undesirable to interrupt execution of $\mathcal{P}\mathsf{rv}$'s application process to obtain a measurement. This is particularly the case for time-sensitive or safety-critical applications. As discussed in Section 7.4, measurements can take non-negligible time, e.g., 7 seconds on an 8-MHz device with 10KB RAM. Making $\mathcal{P}\mathsf{rv}$ unavailable for that long is not appropriate. As is, pure on-demand attestation is a poor match for such applications. At the same time, if $\mathcal{P}\mathsf{rv}$ follows a strict schedule, ERASMUS is also not a remedy since it suffers from the same issue. However, it can be made more flexible.

One partial measure is for $\mathcal{P}\mathsf{rv}$ to be self-aware of when time-sensitive tasks occur. That way, it can schedule measurements at appropriate times. If this knowledge is also available to $\mathcal{V}\mathsf{rf}$, on-demand attestation could be used if $\mathcal{V}\mathsf{rf}$ adapts to $\mathcal{P}\mathsf{rv}$'s schedule.

We consider another ERASMUS variant: *lenient scheduling*. In it, $\mathcal{P}\mathsf{rv}$ is allowed to abort the measurement in progress. If something causes a measurement to be aborted, it can be rescheduled to the end of the current measurement window. However, this comes with some caveats: First, the security architecture needs to be adapted to allow interrupts during measurements. Protection of keys (and cleanup, in case of an interrupt) is still required. Thus, there is still a need for some hardware support. Second, it would be trivial for malware to abort computation of measurements in order to avoid detection, or simply pretend, when queried by $\mathcal{V}\mathsf{rf}$, that all attempted measurements have been aborted. Therefore, $\mathcal{V}\mathsf{rf}$ must use some external information or policy to decide whether there is a valid justification for each aborted measurement. For instance, $\mathcal{V}\mathsf{rf}$ can consider that a maximum of w consecutive missing measurements are acceptable. More than that might indicate that a malware purposely aborted measurements to remain undetected. The value of w represents an evident security/availability tradeoff.

These are certainly not ideal measures and the underlying problem seems quite difficult to address deterministically. As is typical for security/usability compromises, real deployment would likely involve policy-based decisions.

7.6 Collective Remote Attestation (cRA)

Some applications require attesting a group (or swarm) of interconnected embedded devices. In such a setting, it is beneficial to take advantage of inter-connectivity and perform collective attestation using a dedicated protocol. Several cRA techniques have been proposed. SEDA [7] is the first such scheme, which relies on hybrid attestation security architectures: SMART [31] and TrustLite [47]. SEDA combines them with a request-flooding and response-gathering protocol. SEDA was improved and further specified in LISA [16]. Other related techniques deal with report aggregation [76] or physical attacks [38].

ERASMUS could be used instead of on-demand attestation in the context of cRA protocols. In particular, $\mathcal{P}\mathsf{rv}$ self-measurements can be coupled with a collection protocol, such as LISA-α, where the latter only relays reports and does not perform any computation. This would yield a clean and conceptually simple approach to cRA, with all the benefits of ERASMUS.

An additional advantage of using ERASMUS in the swarm setting is support for high mobility. Prior cRA techniques, such as SEDA or SANA require swarm topology to remain almost static during the whole cRA instance. This process may be long and prohibitive for applications where connectivity changes often. ERASMUS does not require external input and its collection phase is very fast, since it does not involve any computation; only reading and sending stored measurements. This makes ERASMUS a very natural and viable technique for highly-mobile swarms.

Finally, related to the discussion in Section 7.5, we consider the scenario where availability of at least one in (or a part of) a group of devices is required at all times. This cannot be guaranteed by on-demand cRA, where a large part of the network may be concurrently busy. Meanwhile, with ERASMUS, it is trivial to establish a schedule which ensures that only a fraction of the swarm computes measurements at any given time.

Chapter 8

Group Consideration

8.1 Introduction

The number of so-called Internet of Things (IoT) devices is expected to soon [14] exceed that of traditional computing devices, i.e., PCs, laptops, tablets and smartphones. IoT can be loosely defined as a set of interconnected embedded devices, each with a various blend of sensing, actuating and computing capabilities. In many IoT settings and use-cases, devices operate collectively as part of a group or swarm, in order to efficiently exchange information and/or collaborate on common tasks. Examples of IoT swarms include multitudes of interconnected devices in smart environments, such as a *smart* households, factories, and buildings. Actual devices might include home theater sound systems, home camera and surveillance systems, electrical outlets, light fixtures, sprinklers, smoke/CO_2 detectors, faucets, appliances, assembly-line components as well as drones. Device swarms also appear in agriculture, e.g., livestock monitoring [70], as well as other research areas, e.g., swarm robotics and swarm intelligence [77]. As IoT swarms become increasingly realistic, their security and overall well-being becomes both apparent and important. Specifically, it is necessary to periodically (or on demand) ensure collective integrity of software running on swarm devices.

The formidable impact of large-scale remote malware infestations has been initially demonstrated by the Stuxnet incident in 2011 and the most recent Dyn denial-of-service (DoS) attack in 2016. This attack type aims to compromise as many devices as possible, without physical access, or close proximity, to any victim device. Compromise of “smart” household devices may also have significant privacy ramifications. In one recent incident, cameras in compromised smart TVs were used to record private activities of their owners [88]. It is not hard to imagine other such attacks, e.g., malware that performs physical DoS by activating smart door locks, sprinklers, or light-bulbs.

8.1.1 Collective Remote Attestation (cRA)

Both feasibility and efficacy of hybrid remote attestation approaches[1] have been demonstrated in the single-prover scenario (See Chapter 2 for more details). Nonetheless, new issues emerge when it is necessary to attest a potentially large number (group or swarm) of devices. First, it is inefficient and sometimes impractical to naïvely apply single-prover remote attestation techniques to each device in a large swarm that might cover a large geographical area. Second, cRA needs to take into account topology discovery, key management and routing. This can be further complicated by mobility (i.e., dynamic topology) and device heterogeneity, in terms of computing and communication resources.

A recently proposed scheme, Scalable Embedded Device Attestation (SEDA) [7], represents the first step towards practical cRA. It builds upon hybrid SMART and TrustLite techniques. It combines them with a flooding-like protocol that propagates attestation requests and gathers corresponding replies. According to simulations in [7], SEDA performs significantly

[1] In the context of remote attestation, we use the following terms interchangeably throughout the chapter: protocols, techniques, methods and approaches.

better than individually attesting each device in a swarm. Despite its viability as a paper design, SEDA is not a practical technique, for several reasons. First, it is under-specified in terms of:

- **Architectural Impact:** What is the impact of cRA on the underlying hardware and security architecture (which suffices for single-prover settings), in terms of: (a) additional required features, as well as (b) increased size and complexity of current features?
- **Timing:** How to determine overall attestation timeout for the verifier? This issue is not as trivial as it might seem, as we discuss later in the chapter.
- **Initiator Selection:** How to select the device(s) that start(s) the attestation process in order to construct a spanning tree over the swarm topology?

Second, as we discuss later, SEDA has some gratuitous (unnecessary) features, such as the use of public key cryptography, which are unjustified by the assumed attack model. Third, it is unclear whether SEDA handles device (node) mobility. This is an important issue: some swarm settings are static in nature, while others involve node mobility and dynamic topologies.

Finally, SEDA does not capture or specify the exact quality of the overall attestation outcome and thus provides no means to compare its security guarantees to other swarm attestation techniques. We believe that it is important to define a qualitative (and whenever possible, quantitative) measure for cRA, i.e., *Quality of Swarm Attestation* (QoSA). This measure should reflect verifier's information requirements and allow comparisons across cRA techniques.

8.2 Preliminaries

We now delineate the scope of our work in this chapter and outline our assumptions.

8.2.1 Scope

This chapter focuses on cRA in the presence of *limited* device mobility, which means that swarm topology is assumed to be connected and quasi-static during each attestation session. The latter means that the swarm connectivity graph can change as long as changes do not influence message propagation during an attestation session.

Similar to prior results in the single-prover remote attestation setting, proposed protocols are not resistant to physical attacks. Other than imposing ubiquitous tamper-resistant hardware, the only practical means of mitigating physical attacks is by heartbeat-based absence detection [38]. We consider this to be an orthogonal direction and focus on remote malware attacks. Also, low-level denial-of-service (DoS) attacks that focus on preventing communication (e.g., physical-layer jamming) are beyond the scope of this chapter. However, we do take into account DoS attacks that try to force devices to perform a lot of computation and/or communication.

Since we build upon state-of-the-art hybrid techniques for single-prover remote attestation, our protocols assume that each device adheres to the minimal requirements specified in [31] and [12]. Even though practicality, i.e., suitability for real-world deployment, is the ultimate goal of this work, we do not actually deploy proposed techniques in real-world swarm settings. Nevertheless, we achieve the next best thing by implementing and evaluating them via emulation, which effectively replicates the behaviors of physical, link and network layers (by virtualizing them on top of Linux) in a virtual environment which takes into account wireless channel interference, noise and loss. Emulation allows us to easily experiment with multiple deployment configurations (varying number of devices, their wireless capabilities and environments) and swarm topologies – something not easily doable in an actual deployed swarm. We claim that, though not the same as actual deployment, emulation is much more realistic than simulation. Since the latter completely abstracts away the protocol stack, it can miss some practical performance issues and artifacts that arise in using the actual stack, the medium access protocol (at the data link layer) and characteristics of the wireless channel (at the physical layer).

8.2.2 Network & Device Assumptions

Devices: We assume that each swarm device (prover):

- Adheres to SMART+ architecture, as discussed in Section 8.2.3 below.
- Has at least one network interface and ability to send/receive both unicast and broadcast packets.
- The second protocol (***LISA****s*) in Section 8.3.2, requires each device to have a clock in order to implement a timer and to know the total number of devices in the swarm – n.

In general, devices can vary along three dimensions: (1) attestation architecture, (2) computational power, and (3) installed code-base. As mentioned above, we assume uniform adherence to SMART+ architecture. Our first protocol, ***LISA***α, makes no other assumptions. The second, ***LISA****s*, also assumes homogeneity in terms of computational power.

Connectivity & Topology: The verifier ($\mathcal{V}\mathsf{rf}$) is assumed to be unaware of the current swarm topology. The topology (connectivity graph) of the swarm can change arbitrarily between any two attestation instances. It might change for a number of reasons, e.g., physical movement of devices, foreign objects impeding or enabling connections between devices, hibernating devices, or devices entering or leaving the network. However, during each attestation instance, the swarm is assumed to be: (1) connected, i.e., there is a path between any pair of devices, and (2) quasi-static. The latter means that the swarm connectivity graph **can** actually change during an attestation session, as long as changes do not influence message propagation, e.g., if a link disappears after one device finishes sending a message, and re-appears before any other message is exchanged between the same pair of devices. See Sections 27 and 32 for details.

If either condition does not hold, protocols discussed in Section 8.3 still provide best-effort attestation, i.e., if a change of connectivity occurs, some healthy devices might end up not being attested, which would result in a false-negative outcome. Nonetheless, infected devices are never positively attested, regardless of any connectivity changes during attestation.

Adversary Type: Based on the recently proposed taxonomy [2], adversaries in the context of remote attestation can be categorized as follows:

- *Remote:* exploits vulnerabilities in prover's software to remotely inject malware. In particular, it can modify any existing code, introduce new malicious code, or read any unprotected memory location.
- *Local:* located sufficiently near the prover to eavesdrop on, and manipulate, prover's communication channels.
- *Physical Non-Intrusive:* located physically near the prover; can perform side-channel attacks in order to learn secrets.
- *Stealthy Physical Intrusive:* can physically compromise a prover; however, it leaves no trace, i.e., can read any (even protected) memory and extract secrets.
- *Physical Intrusive:* has full (local) physical access to the prover; can learn or modify any state of hardware or software components.

Our protocols take into account remote and local adversary flavors. However, as with most prior work, all types of physical attacks are out of scope.

Since all hybrid remote attestation schemes involve hardware-protected key storage, we assume (for now) a trivial master key approach, whereby all swarm devices have the same secret, shared with $\mathcal{V}\mathsf{rf}$. While this might seem silly, it is sufficient in a setting without physical attacks; see Section 8.3 for more details. However, for the sake of completeness, counter-measures to physical attacks and additional cryptographic considerations are discussed in Section 8.6.

8.2.3 Security Architecture

Architectural minimality is a key goal of this work; hence, our protocols require minimal hardware support. Specifically, we assume that each device adheres to the SMART architecture [31], augmented with $\mathcal{V}\mathsf{rf}$ authentication (aka DoS mitigation) features identified in [12]. We refer to this combination as SMART+ and its key aspects are:

- All attestation code (***AttCode***) resides in ROM. ***AttCode*** is safe, i.e., it always terminates and leaks no information beyond the attestation result, i.e., a token.
- Execution of ***AttCode*** is atomic and complete, which means: (a) it can not be interrupted, and (b) it starts and ends at official entry and exit points, respectively.[2] This feature is generally enforced by a Memory Protection Unit (MPU) using a set of static rules.
- At least one secret key stored in ROM, which can only be read from within ***AttCode***. For now, we remain agnostic as far as what type of cryptography is being used.
- A fixed-size block of secure RAM that stores the counter and/or a timestamp of the last executed attestation instance. (This is needed to prevent replay attacks). This memory can only be modified from within ***AttCode***. [12] offers an alternative in the form of a reliable real-time clock that can not be modified by non-physical means. However, we opt for a secure counter since it is a cheaper feature.

[2]There might be multiple legal exit points.

SMART+ operates as follows:

1. $\mathcal{V}$rf generates an authenticated attestation request. Authentication is achieved either via a signature or a message authentication code (MAC), depending on the type of cryptography used.
2. On $\mathcal{P}$rv, the attestation request is received by untrusted code outside ***AttCode*** and passed on to ***AttCode***.
3. ***AttCode*** disables interrupts and checks whether the sequence number of the request is greater than current counter value. If not, request is ignored.
4. ***AttCode*** authenticates – using either symmetric or public key – the attestation request. If authentication fails, request is ignored.
5. ***AttCode*** computes the authenticated integrity check of its memory (i.e., the result), stores it in a publicly accessible location, cleans up after itself, enables interrupts and terminates.
6. Untrusted code on $\mathcal{P}$rv (outside of ***AttCode***) returns the result to $\mathcal{V}$rf.
7. $\mathcal{V}$rf authenticates the result and decides whether $\mathcal{P}$rv is in a secure state.

Memory organization and memory access rules for SMART+ and ***LISA*** are summarized in Figure 8.1.

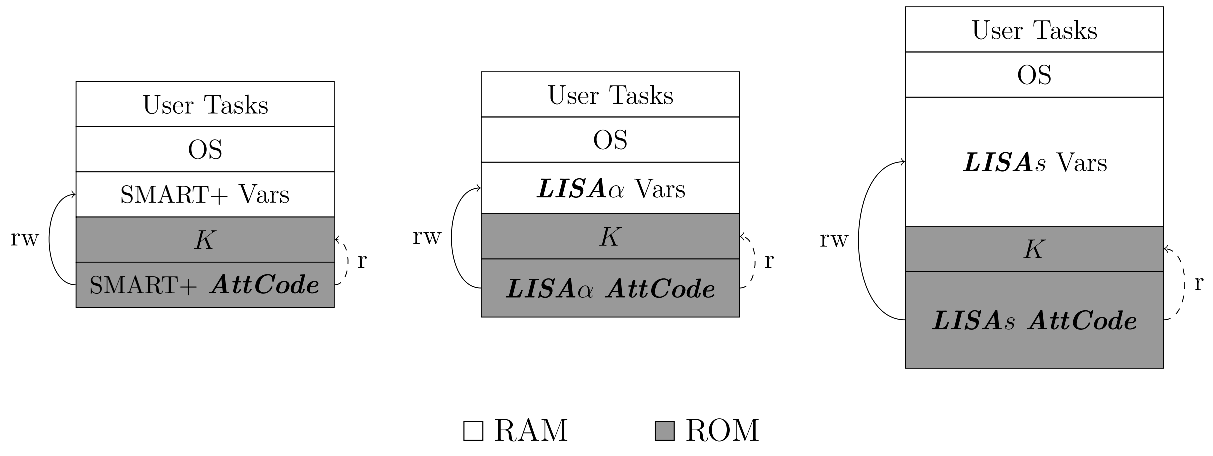


Figure 8.1: Memory organization and access rules in SMART+ and LISA; r denotes read, w denotes write.

8.2.4 Quality of Swarm Attestation (QoSA)

The main goal of cRA is to verify collective integrity of the swarm, i.e., all devices at once. However, in some settings, e.g., when a swarm covers a large physical area, the granularity of a simple binary outcome is not enough. Instead, it might be more useful to learn which devices are potentially infected, so that quick action can be taken to fix them. By the same token, it could be also useful to learn the topology. To this end, we introduce a notion that tries to capture the information provided by swarm attestation, called *Quality of Swarm Attestation (QoSA)*. It also enables comparing multiple cRA protocols. We consider the following types of QoSA:

- *Binary QoSA (B-QoSA):* a single bit indicating success or failure of attestation of the entire swarm.
- *List QoSA (L-QoSA):* a list of identifiers (e.g., link-layer and/or network-layer addresses) of devices that have successfully attested.
- *Intermediate QoSA (I-QoSA):* information that falls between B-QoSA and L-QoSA, e.g., a count of successfully attested devices.
- *Full QoSA (F-QoSA):* a list of attested devices along with their connectivity, i.e., swarm topology.

This is not an exhaustive list. Although we view these four types as fairly natural, other QoSA-s can be envisioned. We also note that, in a single-prover setting which applies to most prior attestation literature, QoSA is irrelevant, since $\mathcal{V}\mathsf{rf}$ communicates directly with one $\mathcal{P}\mathsf{rv}$, and there is no additional information beyond the attestation result itself. In contrast, in a multi-prover setting, QoSA is both natural and useful. It can be tailored to the specific application's needs, as described below in Section 8.3.

8.2.5 Attestation Timeouts

Since envisaged cRA is mostly autonomous and $\mathcal{V}$rf is initially unaware of the current topology, there needs to be an overall timeout value. As in any one-to-many reliable delivery protocol, timeouts are necessary to account for possible losses of connectivity during attestation, caused by mobility, noisy channels, or excessive collisions, all of which might occur naturally, or be caused by DoS attacks. As usual, the timeout parameter must be selected carefully, since an overly low value would result in frequent false positives, while an overly high one would cause unnecessary delays. In any case, we assume that the timeout is dependent on n – the number of devices in the swarm.

8.2.6 Initiator Selection

To minimize its burden, $\mathcal{V}$rf can initiate the process by directly sending an attestation request to one device in the swarm. We call this device an "initiator". There are several ways to select it, e.g., based on physical proximity, and/or computation power. If $\mathcal{V}$rf has no knowledge about nearby devices, it first needs to perform neighbor discovery (e.g., [43] or [49]) which introduces an extra step in the overall process. Alternatively, $\mathcal{V}$rf can use multiple initiators and skip neighbor discovery by simply broadcasting an attestation request to whichever device(s) can hear it. In that case, all $\mathcal{V}$rf's immediate neighbors become initiators, in parallel. Our protocols are agnostic to this choice and work regardless of how initiators are selected, as long as at least one is picked.

8.2.7 Verifier Assumptions

Following prior work, we assume an honest $\mathcal{V}$rf. In particular, it is not compromised and is trusted to correctly generate all attestation requests, as well as to correctly process all received attestation reports (replies). Also, $\mathcal{V}\mathsf{rf}$ is assumed to know n.

8.3 New cRA Protocols

We now describe two lightweight cRA protocols, $\boldsymbol{LISA}\alpha$ and $\boldsymbol{LISA}s$, including their design rationale, details and complexities. Similar to SMART+, either symmetric or public key cryptography can be used to provide authenticated integrity of protocol messages. However, for the sake of simplicity and efficiency, we describe $\boldsymbol{LISA}\alpha$ and $\boldsymbol{LISA}s$ assuming a single swarm-wide symmetric master key. This master key can be pre-installed into all swarm devices at manufacture or deployment time. Although this might seem naïve, recall that, in the absence of physical attacks, there is no difference between having: (1) one swarm-wide master key shared with $\mathcal{V}\mathsf{rf}$, (2) a symmetric unique key each device shares with $\mathcal{V}\mathsf{rf}$, or (3) a device unique public/private key-pair for each device. This is because malware that infects any number of devices still can not access a device's secret key due to SMART+'s MPU access rules. However, if physical attacks are considered, Section 8.6 discusses the use of device-specific symmetric keys and public key cryptography.

8.3.1 Asynchronous Version: $\boldsymbol{LISA}\alpha$

$\boldsymbol{LISA}\alpha$ stands for: **Li**ghtweight **S**warm **A**ttestation, **a**synchronous version. Its goal is to provide efficient cRA while incurring minimal changes over SMART+. Before describing $\boldsymbol{LISA}\alpha$, we can imagine a very intuitive approach, whereby $\mathcal{V}\mathsf{rf}$, relying strictly on SMART+, runs an individual attestation protocol directly with each swarm device. This would require no extra support in terms of software or hardware features. Nonetheless, this naïve approach does not scale, since it requires $\mathcal{V}\mathsf{rf}$ to either: (1) attest each device in sequence, which can be very time-consuming, or (2) broadcast to all devices and maintain state for each, while waiting for replies. This scalability issue motivates device collaboration for propagating attestation requests and reports. $\boldsymbol{LISA}\alpha$ adopts this approach and involves very low computational overhead, while being resistant to computational denial-of-service (DoS) attacks. Devices act independently and asynchronously, relying on each other only for forwarding attestation requests and reports.

$\boldsymbol{LISA}\alpha$ Protocol Details

$\boldsymbol{LISA}\alpha$'s pseudo-code and finite state machine (FSM) for a prover device ($\mathcal{P}\mathsf{rv}$) are illustrated in Algorithm 2 and an upper figure of Figure 8.2, respectively. $\boldsymbol{LISA}\alpha$'s FSM for $\mathcal{V}\mathsf{rf}$ is illustrated in a lower figure of Figure 8.2 and the pseudo-code is described in Algorithm 3. The protocol involves two message types:
(1) *request*: $\mathcal{A}tt_{req} = [\text{“}req\text{”}, \mathcal{S}nd, \mathcal{S}eq, \mathcal{A}uth_{req}]$ and
(2) *report*: $\mathcal{A}tt_{rep} = [\text{“}rep\text{”}, \mathcal{D}evID, \mathcal{P}ar, \mathcal{S}eq, H(\mathcal{M}em), \mathcal{A}uth_{rep}]$
where:

- $\mathcal{S}nd$ – identifier of the sending device; this field is not authenticated
- $\mathcal{S}eq$ – sequence number and/or timestamp of the present attestation instance; set by $\mathcal{V}\mathsf{rf}$
- $\mathcal{A}uth_{req}$ – authentication token for the attestation request message; computed by $\mathcal{V}\mathsf{rf}$ as: $\mathsf{MAC}(K, \text{“}req\text{”}||\mathcal{S}eq)$
- $\mathcal{D}evID$ – identifier of $\mathcal{P}\mathsf{rv}$; stored in ROM, along with ***AttCode***
- $\mathcal{P}ar$ – identifier of the reporting device's parent in the spanning tree; copied from $\mathcal{S}nd$ field in $\mathcal{A}tt_{req}$
- $\mathcal{A}uth_{rep}$ – authentication token for the attestation reply message; computed by $\mathcal{P}\mathsf{rv}$ as: $\mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{D}evID||\mathcal{S}eq||H(\mathcal{M}em))$, where $H()$ is a suitable cryptographic hash function and $\mathcal{M}em$ denotes device memory that is subject to attestation[3].

[3]Note that $H(\mathcal{M}em)$ is part of $\mathcal{A}tt_{rep}$. We can omit it to save space, and have $\mathcal{V}\mathsf{rf}$ keep a mapping of

***LISA*α Prover** From the perspective of a prover $\mathcal{P}\mathsf{rv}$, ***LISA*α** has five states:

1. Wait: $\mathcal{P}\mathsf{rv}$ waits for an attestation-relevant packet. In case of $\mathcal{A}tt_{req}$, $\mathcal{P}\mathsf{rv}$ proceeds to VerifyRequest and in case of $\mathcal{A}tt_{rep}$, it jumps to VerifySession.

2. VerifyRequest: $\mathcal{P}\mathsf{rv}$ first checks validity of $\mathcal{S}eq$, which must be strictly greater than the previous stored value; otherwise, it discards $\mathcal{A}tt_{req}$ and returns to Wait. Next, $\mathcal{P}\mathsf{rv}$ validates $\mathcal{A}uth_{req}$ by recomputing MAC. If verification fails, $\mathcal{P}\mathsf{rv}$ discards $\mathcal{A}tt_{req}$ and returns to Wait. Otherwise, $\mathcal{P}\mathsf{rv}$ saves $\mathcal{S}eq$ as the current session number $\mathcal{C}urSeq$, stores $\mathcal{S}nd$ as its parent device $\mathcal{P}ar$ for this session, and transitions to Attest.

3. Attest: $\mathcal{P}\mathsf{rv}$ sets $\mathcal{S}nd$ field in $\mathcal{A}tt_{req}$ to $\mathcal{D}evID$ and broadcasts the modified $\mathcal{A}tt_{req}$. Next, $\mathcal{P}\mathsf{rv}$ computes $\mathcal{A}uth_{rep}$ and composes $\mathcal{A}tt_{rep}$, as defined above. Note that $\mathcal{A}uth_{rep}$ authenticates $\mathcal{P}ar$ by virtue of covering $\mathcal{A}tt_{req}$. Finally, $\mathcal{P}\mathsf{rv}$ unicasts $\mathcal{A}tt_{rep}$ to $\mathcal{P}ar$ and transitions to Wait.

4. VerifySession: $\mathcal{P}\mathsf{rv}$ receives $\mathcal{A}tt_{rep}$ from one of its descendants. If the $\mathcal{S}eq$ in $\mathcal{A}tt_{rep}$ does not match its stored counterpart $\mathcal{C}urSeq$, $\mathcal{P}\mathsf{rv}$ discards $\mathcal{A}tt_{rep}$ and returns to Wait. Otherwise, it proceeds to Forward.

5. Forward: $\mathcal{P}\mathsf{rv}$ unicasts $\mathcal{A}tt_{rep}$ received in VerifySession, to its stored $\mathcal{P}ar$ and returns to Wait.

$(\mathcal{D}evID, H(\mathcal{M}em))$. However, this would take away $\mathcal{V}\mathsf{rf}$'s ability to make decisions based on actual device signatures.

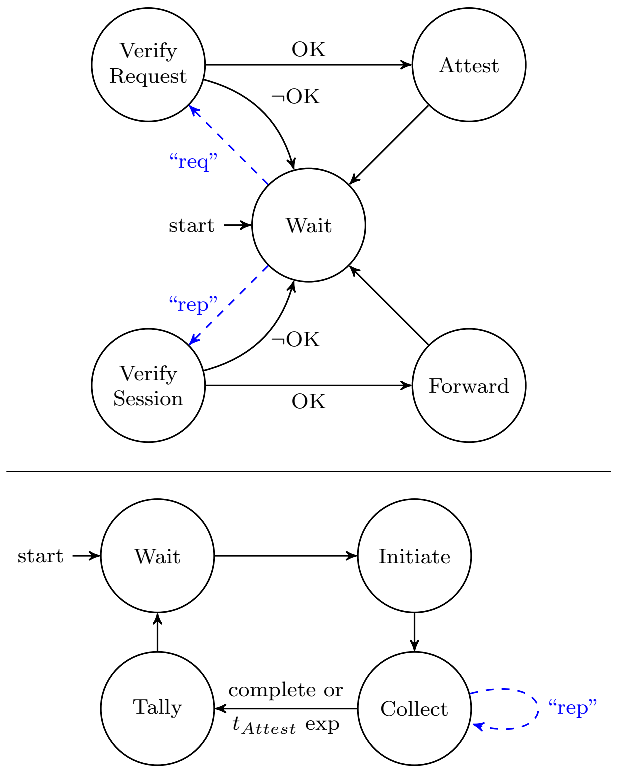


Figure 8.2: ***LISA***α FSM-s for $\mathcal{P}$rv (top) and $\mathcal{V}$rf (bottom)

LISAα **Verifier** From $\mathcal{V}$rf's perspective, ***LISA***α is simpler, with four states:

1. Wait: $\mathcal{V}$rf waits for external signal (e.g., from a user) to start a new attestation session. When it arrives, $\mathcal{V}$rf moves to Initiate.

2. Initiate: $\mathcal{V}$rf sets the overall timeout and selects the initiator(s), as discussed earlier. It then initializes $Attest = Fail = \emptyset$, $Norep = \{all\ \mathcal{D}evID\}$. Next, $\mathcal{V}$rf sets $\mathcal{S}nd$ =$\mathcal{V}$rf, composes $\mathcal{A}tt_{req}$, sends it (via unicast) to the initiator(s), and moves to Collect.

3. Collect: $\mathcal{V}$rf waits for $\mathcal{A}tt_{rep}$ messages from the initiator(s) or an overall timeout. If a timeout occurs, $\mathcal{V}$rf transitions to Tally. Upon receipt of $\mathcal{A}tt_{rep}$, $\mathcal{V}$rf extracts and validates $\mathcal{A}uth_{rep}$ by recomputing MAC. (Note that duplicate $\mathcal{A}tt_{rep}$ messages are assumed to be automatically detected and suppressed). There are three possible outcomes:

i. Validation fails: $\mathcal{A}tt_{rep}$ is discarded,

Algorithm 2: Pseudo-code of $\boldsymbol{LISA}\alpha$ for $\mathcal{P}$rv

```
Write-Protected Vars: $\mathcal{D}evID$ – id of $\mathcal{P}$rv
                      $\mathcal{C}urSeq$ – current sequence #
                      $\mathcal{P}ar$ – $\mathcal{P}$rv's parent id
while $True$ do
    $m$ = RECEIVE();
    if TYPE($m$) = "$req$" then
        [$\mathcal{S}nd$, $\mathcal{A}uth_{req}$, $\mathcal{S}eq$] $\leftarrow$ DECOMPOSE($m$);
        if $\mathcal{S}eq < \mathcal{C}urSeq$ then
            CONTINUE();
        end
        if $\mathcal{A}uth_{req} \neq \mathsf{MAC}(K, \text{“}req\text{”}||\mathcal{S}eq)$ then
            CONTINUE();
        end
        $\mathcal{C}urSeq \leftarrow \mathcal{S}eq$; $\mathcal{P}ar \leftarrow \mathcal{S}nd$;
        BROADCAST("$req$"$||\mathcal{D}evID||\mathcal{C}urSeq||\mathcal{A}uth_{req}$);
        $\mathcal{A}uth_{rep} \leftarrow \mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{D}evID||\mathcal{C}urSeq||H(\mathcal{M}em))$;
        $\mathcal{A}tt_{rep} \leftarrow \text{“}rep\text{”}||\mathcal{D}evID||\mathcal{P}ar||\mathcal{C}urSeq||H(\mathcal{M}em)||\mathcal{A}uth_{rep}$;
        UNICAST($\mathcal{P}ar, \mathcal{A}tt_{rep}$);
    else if TYPE($m$) = "$Rep$" then
        $\mathcal{S}eq \leftarrow$ GETSEQ($m$);
        if $\mathcal{S}eq = \mathcal{C}urSeq$ then
            UNICAST($\mathcal{P}ar, m$);
        end
    end
end
```

ii. $\mathcal{A}uth_{rep}$ is authentic and $H(\mathcal{M}em)$ corresponds to an expected (legal) state of $\mathcal{D}evID$'s attested memory: $\mathcal{V}$rf adds $\mathcal{D}evID$ to $Attest$, and removes it from $Norep$,

iii. $\mathcal{A}uth_{rep}$ is authentic and $H(\mathcal{M}em)$ does not match any expected state of $\mathcal{D}evID$'s attested memory: $\mathcal{V}$rf adds $\mathcal{D}evID$ to $Fail$ and removes it from $Norep$

If $|Attest| + |Fail| = n$, $\mathcal{V}$rf moves to Tally; otherwise it remains in Collect.

4. Tally: $\mathcal{V}$rf outputs $Attest$, $Fail$ and $Norep$ as sets of devices that passed, failed and didn't reply, respectively. Finally, $\mathcal{V}$rf returns to Wait.

Algorithm 3: Pseudo-code of ***LISA***α for $\mathcal{V}$rf

```
$t_{Attest} \leftarrow t_a + t_{MAC} + 2 \cdot n \cdot t_t + t_s$;
while $True$ do
    $wait()$;
    $\mathcal{I}nitID \leftarrow$ getInitId();
    $\mathcal{C}urSeq \leftarrow$ getSeq();
    $\mathcal{A}tt_{req} \leftarrow$ "req"$||\mathcal{V}\mathsf{rf}||\mathcal{C}urSeq||\mathcal{A}uth_{req}$;
    unicast($\mathcal{I}nitID, \mathcal{A}tt_{req}$);
    $Attest \leftarrow \emptyset$; $Fail \leftarrow \emptyset$;
    $Norep \leftarrow \{all\mathcal{D}evID\}$;
    $T \leftarrow$ getTimer();
    while $T < t_{Attest}$ do
        $\mathcal{A}tt_{rep} \leftarrow$ receive();
        $[\mathcal{D}evID, \mathcal{P}ar, \mathcal{S}eq, H(\mathcal{M}em), \mathcal{A}uth_{rep}] \leftarrow$ decompose($\mathcal{A}tt_{rep}$);
        if $\mathcal{S}eq = \mathcal{C}urSeq \wedge \mathcal{A}uth_{rep} = \mathsf{MAC}(K,$ "rep"$||\mathcal{D}evID||\mathcal{C}urSeq||H(\mathcal{M}em))$ then
            if $H(\mathcal{M}em) \subset$ expectedHash($\mathcal{D}evID$) then
                $Attest \leftarrow Attest \cup \{\mathcal{D}evID\}$;
            else
                $Fail \leftarrow Fail \cup \{\mathcal{D}evID\}$;
            end
            $Norep \leftarrow Norep \setminus \{\mathcal{D}evID\}$;
        end
        if $|Attest| + |Fail| = n$ then
            break();
        end
    end
    output($Attest, Fail, Norep$);
end
```

$\mathcal{V}$rf Timeout in ***LISA***α

As follows from the protocol description (or, equivalently, from FSMs and pseudocode), devices do not require a timeout. For its part, $\mathcal{V}$rf sets the overall attestation timeout to $t_{Attest} = t_a + n \cdot t_{MAC} + 2 \cdot n \cdot t_t + t_s$, **where:**

- t_a – time for $\mathcal{P}$rv to perform self-attestation[4]
- t_{MAC} – time for $\mathcal{P}$rv to compute a MAC (to verify or generate) over a short message
- t_t – time for $\mathcal{P}$rv to transmit a message to another device
- t_s – slack time, which accounts for variabilities, i.e., possible deviations

t_{Attest} represents the time corresponding to running ***LISA***α over a n-device swarm with the worst-case topology scenario, i.e., a realistic upper bound. The worst-case is a **line topology** where $\mathcal{A}tt_{req}$ processing is done in sequence, taking $n \cdot t_{MAC}$. Only one t_a needs

[4]In case of heterogeneous devices, t_a represents the maximum self-attestation time across all devices. The same applies to t_{MAC} and t_t.

to be included in t_{Attest} since the last device (the only leaf in the tree) finishes its attestation after all others. Also, since there are at most n hops between $\mathcal{V}\mathsf{rf}$ and the last device, it takes $n \cdot t_t$ to transmit $\mathcal{A}tt_{req}$ to that device and the same amount of time to transmit the last $\mathcal{A}tt_{rep}$ to $\mathcal{V}\mathsf{rf}$.

Connectivity in *LISA*α

Let t_0 denote the time when $\mathcal{P}\mathsf{rv}$ receives $\mathcal{A}uth_{req}$ from $\mathcal{P}ar$, $t_{rep,i}$ denote the time when $\mathcal{P}ar$ receives the i^{th} $\mathcal{A}uth_{rep}$ from $\mathcal{P}\mathsf{rv}$ and z denote the number of $\mathcal{P}\mathsf{rv}$'s descendants. The connectivity assumption of ***LISA***α can be formally stated as follows:

LISAα produces a correct cRA result, i.e. no false positives and no false negatives, if a link between every $\mathcal{P}\mathsf{rv}$ and its $\mathcal{P}ar$ exists during their t_0, $t_{rep,1}$, $t_{rep,2}$,...,$t_{rep,z+1}$.

QoSA of *LISA*α

At the end, $\mathcal{V}\mathsf{rf}$ collects a set of $\mathcal{A}tt_{rep}$ messages, one from each device. After verifying all $\mathcal{A}tt_{rep}$-s, $\mathcal{V}\mathsf{rf}$ learns the list of successfully attested devices, thus achieving L-QoSA. It is easy to augment the protocol to collect topology information along with attestation results. This can be performed by simply including $\mathcal{P}ar$ in each $\mathcal{A}tt_{rep}$. $\mathcal{V}\mathsf{rf}$ then can thus reconstruct the topology based on verified reports. Specifically, line 15 in Algorithm 2 would become:

$\mathcal{A}uth_{rep} \leftarrow \mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{P}ar||H(\mathcal{M}em))$;

However, topology information obtained by $\mathcal{V}\mathsf{rf}$ is not reliable, since $\mathcal{P}ar$ is not authenticated upon receiving $\mathcal{A}tt_{req}$. Fixing this is not hard; it would require each device to: (1) compute and attach an extra MAC, at least over $\mathcal{P}ar$ and $\mathcal{A}uth_{req}$ fields, at $\mathcal{A}tt_{req}$ forwarding time, and (2) verify the $\mathcal{P}ar$'s MAC upon receiving $\mathcal{A}tt_{req}$.

Complexity of $\boldsymbol{LISA}\alpha$

Architectural Impact: Roughly speaking, the $\boldsymbol{LISA}\alpha$ protocol adheres to SMART+ security architecture, i.e., it does not impose any additional features. However, it clearly requires a larger ROM to house additional code, and a more complex MPU to implement access rules. Compared to SMART+, ROM size is expected to grow by just 30 bytes, as shown in Table 8.1. Also, $\boldsymbol{LISA}\alpha$ introduces two extra write-protected variables: $\mathcal{S}eq$ and $\mathcal{P}ar$. We assume that each can be a 32-bit value, i.e., only 8 extra bytes need to be write-protected. Finally, MPU needs to support two additional access rules to protect these two variables. The resulting increase in hardware complexity is shown in Figure 8.1 and Table 8.1.

Table 8.1: Estimated code complexity; all code in "C".

	METHOD:			
	SMART+ w/o MAC[5]	SMART+	$\boldsymbol{LISA}\alpha$	$\boldsymbol{LISA}s$
Lines of Code	43	262	269	321
Executable Size (bytes)	8,565	17,896	17,928	18,128
Write-Protected Memory Size (bytes)	n/a	4	12	$40 + 4n$

Software Complexity: $\boldsymbol{LISA}\alpha$ needs only one simple extra operation: message ($\mathcal{A}tt_{req}$) broadcast. This operation is usually straight-forward in practice if a device is already capable of unicasting. Moreover, $\boldsymbol{LISA}\alpha$ is nearly stateless: only $\mathcal{S}eq$ and $\mathcal{P}ar$ need to persist between attestation sessions. Table 8.1 shows that $\boldsymbol{LISA}\alpha$ is only about 2% higher than single-prover SMART+ in terms of lines-of-code (LoC-s).

Communication Overhead: We assume an SHA-256-based hash and MAC constructs, each yielding a 32-byte output. Overall size of $\mathcal{A}tt_{req}$ is thus 43 bytes: 3 – message tag, 4 – $\mathcal{S}nd$, 4 – $\mathcal{S}eq$, and 32 – $\mathcal{A}uth_{req}$. Meanwhile, $\mathcal{A}tt_{rep}$ is 79 bytes: 3 – message tag, 4 – $\mathcal{D}evID$, 4 – $\mathcal{P}ar$, 4 – $\mathcal{S}eq$, 32 – $\mathcal{A}uth_{rep}$, and 32 – $H(\mathcal{M}em)$. We also assume that Prv has

[5]MAC is implemented as HMAC-SHA-256 from [5]

z descendants and w neighbors in the swarm spanning tree, and there are n devices total.

In each session, $\mathcal{P}\mathsf{rv}$ receives up to w $\mathcal{A}tt_{req}$-s and exactly z $\mathcal{A}tt_{rep}$-s. Thus, depending on topology, $\mathcal{P}\mathsf{rv}$ might receive anywhere between $(43 + 79z)$ and $(43w + 79z)$ bytes. Also, $\mathcal{P}\mathsf{rv}$ broadcasts one $\mathcal{A}tt_{req}$ to neighbors and unicasts $(z + 1)$ $\mathcal{A}tt_{rep}$-s to $\mathcal{P}ar$. Thus, overall transmission cost for each $\mathcal{P}\mathsf{rv}$ is: $43 + 79(z + 1)$.

Clearly, potentially high communication overhead is ***LISA***α's biggest drawback, since a device – in the worst case – transmits n reports. This motivated us to design ***LISA****s* which reduces communication overhead by aggregating $\mathcal{A}tt_{rep}$-s.

8.3.2 Synchronous Version: ***LISA****s*

The main idea in ***LISA****s* is to let devices authenticate and attest each other. When one device is attested by another, only the identifier of the former needs to be securely forwarded to $\mathcal{V}\mathsf{rf}$, instead of the entire $\mathcal{A}tt_{rep}$. This translates into considerable bandwidth savings and lower $\mathcal{V}\mathsf{rf}$ workload. Also, $\mathcal{A}tt_{rep}$-s can be aggregated, which decreases the number of packets sent and received. It also allows more flexibility in terms of QoSA: from B-QoSA to F-QoSA. Finally, malformed or fake $\mathcal{A}tt_{rep}$-s are detected in the network and not propagated to $\mathcal{V}\mathsf{rf}$, as in ***LISA***α. However, these benefits are traded off for increased protocol (and code) complexity, as described below.

LISA*s*'s main distinctive feature is that each $\mathcal{P}\mathsf{rv}$ waits for all of its children'$\mathcal{A}tt_{rep}$-s before submitting its own. This makes the protocol synchronous. Each $\mathcal{P}\mathsf{rv}$ keeps track of its parent and children during an attestation session. Once $\mathcal{A}tt_{req}$ is processed and propagated, $\mathcal{P}\mathsf{rv}$ waits for each child to complete attestation by submitting a $\mathcal{A}tt_{rep}$. Then, $\mathcal{P}\mathsf{rv}$ verifies each $\mathcal{A}tt_{rep}$, aggregates a list of children as well as descendants they attested, attests itself, and finally sends its authenticated $\mathcal{A}tt_{rep}$ (which contains the list of attested descendants) to its

$\mathcal{P}ar$.

LISAs Protocol Details

The FSM and pseudo-code for $\mathcal{P}\mathsf{rv}$ are shown in an upper part of Figure 8.3 and Algorithm 4, respectively. ***LISAs*** is constructed such that $\mathcal{P}\mathsf{rv}$ can receive a new $\mathcal{A}tt_{req}$ in any state, even while waiting for children's $\mathcal{A}tt_{rep}$-s. Besides $\mathcal{A}tt_{req}$ and $\mathcal{A}tt_{rep}$, ***LISAs*** involves one extra message type:

(1) *request*: $\mathcal{A}tt_{req} = [\text{“}req\text{”}, \mathcal{S}nd, \mathcal{S}eq, \mathcal{D}epth, \mathcal{A}uth_{req}]$,
(2) *report*: $\mathcal{A}tt_{rep} = [\text{“}rep\text{”}, \mathcal{S}eq, \mathcal{D}evID, \mathcal{D}esc, \mathcal{A}uth_{rep}]$, and
(3) *acknowledgment*: $\mathcal{A}tt_{ack} = [\text{“}ack\text{”}, \mathcal{S}eq, \mathcal{D}evID, \mathcal{P}ar]$, with:

- $\mathcal{S}nd$– identifier of the sending device; this field is not authenticated
- $\mathcal{S}eq$– sequence number and/or timestamp of the present attestation instance; set by $\mathcal{V}\mathsf{rf}$
- $\mathcal{D}epth$– depth of the sending device in the spanning tree
- $\mathcal{A}uth_{req}$– authentication token for the attestation request message; computed by $\mathcal{V}\mathsf{rf}$ as: $\mathsf{MAC}(K, \text{“}req\text{”}||\mathcal{S}eq)$
- $\mathcal{D}evID$– identifier of $\mathcal{P}\mathsf{rv}$ (in ROM, along with ***AttCode***)
- $\mathcal{D}esc$– list of $\mathcal{P}\mathsf{rv}$'s descendants; populated when $\mathcal{P}\mathsf{rv}$ receives an authentic report
- $\mathcal{A}uth_{rep}$– authentication token for the attestation reply message; computed by $\mathcal{P}\mathsf{rv}$ as: $\mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{S}eq||\mathcal{D}evID||\mathcal{D}esc)$
- $\mathcal{P}ar$– identifier of reporting device's parent in the spanning tree; copied from $\mathcal{S}nd$ field in $\mathcal{A}tt_{req}$

Prover in *LISAs* From the perspective of a prover $\mathcal{P}\mathsf{rv}$, ***LISAs*** consists of eight states:

Algorithm 4: $\mathcal{P}\mathsf{rv}$ pseudo-code in ***LISA****s*.

```
Write-Protected Vars: $\mathcal{C}urSeq$ – current sequence number
                      $\mathcal{D}evID$ – id of Device $\mathcal{P}\mathsf{rv}$
                      $\mathcal{P}ar$ – id of current $\mathcal{P}\mathsf{rv}$'s parent
                      $\mathcal{C}$ – pre-installed hash of $\mathcal{P}\mathsf{rv}$'s memory
                      $\mathcal{D}esc$ – list of id-s of $\mathcal{P}\mathsf{rv}$'s descendants
$t_{ACK} \leftarrow t_{MAC} + 2t_t + t_s$;
while $True$ do
    $m \leftarrow$ NONBLOCKRECEIVE();
    $T \leftarrow$ GETTIME();
    if TYPE($m$) = "$req$" then
        [$\mathcal{A}uth_{req}$, $\mathcal{S}eq$, $\mathcal{S}nd$, $\mathcal{D}epth$ ] $\leftarrow$ DECOMPOSE($m$, "$req$");
        if $\mathcal{S}eq < \mathcal{C}urSeq$ then
            CONTINUE();
        end
        if $\mathcal{A}uth_{req} \neq \mathsf{MAC}_K(\text{"}req\text{"}\|\mathcal{S}eq)$ then
            CONTINUE();
        end
        $\mathcal{C}urSeq \leftarrow \mathcal{S}eq$; $\mathcal{P}ar \leftarrow \mathcal{S}nd$;
        $\mathcal{A}tt_{ack} \leftarrow$ "$ack$"$||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{P}ar$;
        UNICAST($\mathcal{P}ar, \mathcal{A}tt_{ack}$);
        $t_{REP} \leftarrow (n - \mathcal{D}epth)(t_{ACK} + t_a + t_{MAC} + t_t + t_s)$;
        $\mathcal{A}tt_{req} \leftarrow$ "$req$"$||\mathcal{C}urSeq||\mathcal{D}evID||(\mathcal{D}epth + 1)||\mathcal{A}uth_{req}$;
        BROADCAST($\mathcal{A}tt_{req}$);
        $\mathcal{D}esc \leftarrow \emptyset$;
        $\mathcal{C}hildren \leftarrow \emptyset$;
        $T \leftarrow$ RESTARTTIMER();
    else if TYPE($m$) = "$ack$" then
        if $T > t_{ACK}$ then
            CONTINUE();
        end
        [$\mathcal{S}eq$, $\mathcal{S}nd$, $\mathcal{S}ndPar$ ] $\leftarrow$ DECOMPOSE($m$, "$ack$");
        if $\mathcal{S}eq = \mathcal{C}urSeq$ then
            $\mathcal{C}hildren = \mathcal{C}hildren \cup \{\mathcal{S}nd\}$;
        end
    else if TYPE($m$) = "$rep$" then
        if $T \leq t_{ACK} \vee T \geq t_{REP}$ then
            CONTINUE();
        end
        [$\mathcal{S}eq$, $\mathcal{S}nd$, $\mathcal{S}ndDesc$, $\mathcal{A}uth_{rep}$ ] $\leftarrow$ DECOMPOSE($m$, "$rep$");
        if $\mathcal{S}eq \neq \mathcal{C}urSeq$ then
            CONTINUE();
        end
        if $\mathcal{A}uth_{rep} = \mathsf{MAC}(K, \text{"}rep\text{"}||\mathcal{C}urSeq||\mathcal{S}nd||\mathcal{S}ndDesc)$ then
            $\mathcal{D}esc \leftarrow \mathcal{D}esc \cup \{\mathcal{S}nd\} \cup \mathcal{S}ndDesc$;
            $\mathcal{C}hildren \leftarrow \mathcal{C}hildren \setminus \{\mathcal{S}nd\}$;
        end
    end
    if $(\mathcal{C}hildren = \emptyset \wedge T \geq t_{ACK}) \vee (T \geq t_{REP})$ then
        if $H(\mathcal{M}em) \neq \mathcal{C}$ then
            ABORT();
        end
        $\mathcal{A}uth_{rep} \leftarrow \mathsf{MAC}(K, \text{"}rep\text{"}||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{D}esc)$;
        $\mathcal{A}tt_{rep} \leftarrow$ "$rep$"$||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{D}esc||\mathcal{A}uth_{rep}$;
        UNICAST($\mathcal{P}ar, \mathcal{A}tt_{rep}$);
        $T \leftarrow$ RESETANDSTOPTIMER();
    end
end
```

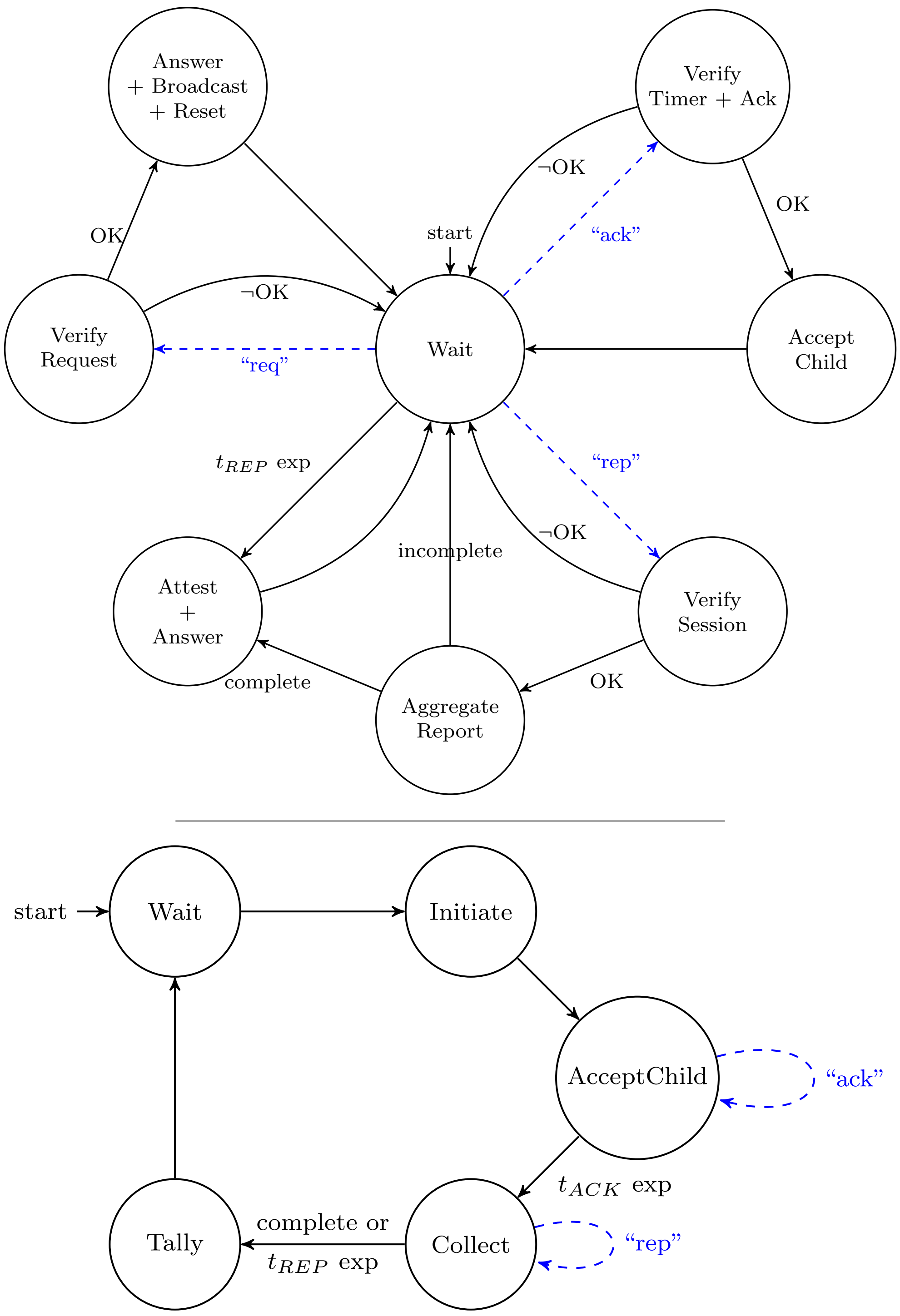


Figure 8.3: FSM of ***LISA****s*: Dev (top) and Vrf (bottom)

1. Wait: the initial state where $\mathcal{P}$rv waits for an attestation-relevant packet. $\mathcal{P}$rv transitions to VerifyRequest if it is $\mathcal{A}tt_{req}$, VerifySession if it is $\mathcal{A}tt_{rep}$ and VerifyTimer+Ack if it is $\mathcal{A}tt_{ack}$. Also, if a timeout occurs during this state, $\mathcal{P}$rv transitions to Attest+Answer. This timeout

is set in Answer+Broadcast+Reset below.

2. VerifyRequest: This state is similar to VerifyRequest in $\boldsymbol{LISA}s$. If verification of $\mathcal{A}tt_{req}$ fails, $\mathcal{P}$rv discards $\mathcal{A}tt_{req}$ and goes back to Wait. Otherwise, $\mathcal{P}$rv realizes its depth in the spanning tree through $\mathcal{D}epth$ field in $\mathcal{A}tt_{req}$ and saves $\mathcal{S}eq$ as $\mathcal{C}urSeq$ and $\mathcal{S}nd$ as $\mathcal{P}ar$. Finally, $\mathcal{P}$rv transitions to Answer+Broadcast+Reset.

3. Answer+Broadcast+Reset: $\mathcal{P}$rv sends $\mathcal{A}tt_{ack}$ back to $\mathcal{P}ar$, copies its $\mathcal{D}evID$ into $\mathcal{S}nd$ field of $\mathcal{A}tt_{req}$ and broadcasts the modified $\mathcal{A}tt_{req}$. Next, $\mathcal{P}$rv computes a timeout t_{REP}. This timeout is used to determine when to stop receiving $\mathcal{A}tt_{rep}$ during Wait. $\mathcal{P}$rv then initializes a list of its children ($\mathcal{C}hildren$) and a list of its descendants ($\mathcal{D}esc$) to empty sets, starts a timer, and returns to Wait.

4. VerifyTimer+Ack: $\mathcal{P}$rv receives $\mathcal{A}tt_{ack}$ from a device that wants to be its child. First, $\mathcal{P}$rv checks with an acknowledgment timeout (t_{ACK}), which is a global constant. If the current time is later than t_{ACK}, $\mathcal{P}$rv discards $\mathcal{A}tt_{ack}$ and returns to Wait. If the $\mathcal{S}eq$ in $\mathcal{A}tt_{ack}$ does not match $\mathcal{C}urSeq$, $\mathcal{P}$rv also discards $\mathcal{A}tt_{ack}$ and goes back to Wait. Otherwise, $\mathcal{P}$rv transitions to AcceptChild.

5. AcceptChild: $\mathcal{P}$rv accepts $\mathcal{A}tt_{ack}$ and stores $\mathcal{S}nd$ into $\mathcal{C}hildren$. Then, $\mathcal{P}$rv returns to Wait.

6. VerifySession: This state is also similar to VerifySession in $\boldsymbol{LISA}\alpha$. $\mathcal{P}$rv discards $\mathcal{A}tt_{rep}$ and return to Wait if $\mathcal{S}eq$ in $\mathcal{A}tt_{rep}$ does not match $\mathcal{C}urSeq$. Otherwise, it transitions to AggregateReport

7. AggregateReport: $\mathcal{P}$rv accepts $\mathcal{A}tt_{rep}$ and aggregates it with other received reports in the same session. The aggregation is done by adding $\mathcal{S}nd$ and $\mathcal{D}esc$ fields in $\mathcal{A}tt_{rep}$ into its $\mathcal{D}esc$ and removing $\mathcal{S}nd$ from $\mathcal{C}hildren$ since $\mathcal{S}nd$ has replied. If all of its children have already replied (or $\mathcal{C}hildren = \emptyset$), $\mathcal{P}$rv transitions to Attest+Answer. Otherwise, $\mathcal{P}$rv returns to Wait.

8. Attest+Answer: $\mathcal{P}$rv computes a hash of its attestable memory. If the resulting digest does

not match with the pre-installed hash value ($\mathcal{C}$), $\mathcal{P}\mathsf{rv}$ outputs an error and acts accordingly (e.g., hardware reset and memory wipe-out). Otherwise, $\mathcal{P}\mathsf{rv}$ constructs $\mathcal{A}uth_{rep}$ and $\mathcal{A}tt_{rep}$ as defined earlier and unicasts $\mathcal{A}tt_{rep}$ to $\mathcal{P}ar$. Finally, the timer is reset and stopped and $\mathcal{P}\mathsf{rv}$ returns to Wait.

We now consider some details of Algorithm 4.

- Line 3: Reception of messages should be non-blocking, such that the timer can be checked even when no message is received[6].
- Line 7: Freshness of $\mathcal{S}eq$ in $\mathcal{A}tt_{req}$ is established by comparing it to $\mathcal{C}urSeq$, as in ***LISA***α. During any given session, a node acknowledges to the first neighbor that broadcasts an attestation request with $\mathcal{C}urSeq$. Subsequent broadcasts with the same $\mathcal{C}urSeq$ are ignored.
- Line 14: $\mathcal{A}tt_{ack}$ to $\mathcal{P}ar$ is constructed, consisting of: $\mathcal{S}eq$, $\mathcal{D}evID$ and $\mathcal{P}ar$. These values are not authenticated since $\mathcal{A}tt_{ack}$ is only used for determining timeouts. An adversary can send fake $\mathcal{A}tt_{ack}$-s to $\mathcal{P}\mathsf{rv}$ which would only cause longer timeouts.
- Line 17: The request contains: $\mathcal{S}eq$, $\mathcal{S}nd$ and $\mathcal{D}epth$. Authentication of $\mathcal{S}eq$ is required to prevent replay attacks while $\mathcal{S}nd$ and $\mathcal{D}epth$ do not need to be authenticated. If either or both of the latter are modified by a local adversary, the result would be a DoS attack.
- Lines 19 and 20: The sets $\mathcal{D}esc$ and $\mathcal{C}hildren$ are re-initialized, i.e., set to empty. The former represents $\mathcal{D}evID$'s of $\mathcal{P}\mathsf{rv}$'s descendants, which is populated when reports from children are verified (line 39). The latter represents the set of $\mathcal{P}\mathsf{rv}$'s children. It is populated whenever a neighbor acknowledgment is verified (line 28) and de-populated when a child's report is verified (line 40). If $\mathcal{C}hildren$ is empty at any time after t_{ACK}, it means all the reports of $\mathcal{P}\mathsf{rv}$'s children have been attested and $\mathcal{P}\mathsf{rv}$ may proceed with self-attestation (line 43).

[6]Note that, in loops with only non-blocking operations, it is necessary to avoid busy waiting; this is usually done by adding a short sleep timer at each iteration.

- Line 21: The timer is reset whenever a request is verified and re-broadcast.
- Lines 27 and 35: The received sequence number $\mathcal{S}eq$ is compared to the one stored for the last accepted request – $\mathcal{C}urSeq$. If they differ, the message (acknowledgment or report) is discarded. This comparison incurs a negligible cost while preventing acknowledgments and reports from older sessions being verified when a new attestation session has started. It also mitigates DoS attacks whereby a remote adversary (unaware of the current $\mathcal{S}eq$) sends fake acknowledgments or reports.
- Lines 44 and 45: A hash of specified memory range of $\mathcal{P}$rv is computed and compared to its reference value – $\mathcal{C}$. If they do not match, ***LISA****s* returns an error, performs a hardware re-set and cleans up its memory. $\mathcal{C}$ needs to be write-protected. This can be enforced by a static MPU rule.
- Line 48: $\mathcal{A}uth_{rep}$ contains authenticated fields: $\mathcal{S}eq$, $\mathcal{S}nd$ and $\mathcal{S}ndDesc$. $\mathcal{S}eq$ and $\mathcal{S}nd$ need to be validated to ensure authenticity and prevent replay attacks. $\mathcal{S}ndDesc$ is also authenticated to prevent a man-in-the-middle attacks that might overwrite attestation status of some descendants. Self-attestation must be performed last – after verifying reports of all children. If performed earlier, the protocol becomes vulnerable to a sort of a time-of-check-to-time-of-use (TOCTOU) attack where $\mathcal{P}$rv gets corrupted after performing self-attestation and before sending out the aggregated report.
- Line 50: The timer is reset and stopped when attestation is completed. It is stopped so the condition at line 43 does not hold until a new $\mathcal{A}uth_{req}$ is received.

Verifier in *LISA*s The $\mathcal{V}$rf in ***LISA****s* has one additional state – AcceptChild – while the rest of the states remain similar or the same as the ones in ***LISA***α. $\mathcal{V}$rf's pseudo-code is illustrated below and its finite state machine is in the lower part of Figure 8.3.

<u>1&2. Wait and Initiate</u>: These two state are identical to their counterparts in ***LISA***α.

<u>3. AcceptChild</u>: $\mathcal{V}$rf waits for $\mathcal{A}tt_{ack}$-s from the initiator(s), which are used to determine the

completion of Collect. After a timeout occurs, $\mathcal{V}$rf stops receiving $\mathcal{A}tt_{ack}$-s and transitions to Collect.

4. Collect: This state is also similar to Collect in ***LISA***α except the following three behaviors: One is $\mathcal{V}$rf does not need to check $H(\mathcal{M}em)$ since it is not include in $\mathcal{A}tt_{rep}$. Secondly, $\mathcal{V}$rf does not need to maintain a list of unsuccessfully attested devices ($Fail$) since software-infected devices cannot output an authentic $\mathcal{A}uth_{rep}$. Lastly, $\mathcal{V}$rf transitions to Tally if the initiator(s) (realized in AcceptChild) has sent its reports. The rest of its behaviors remains the same as in ***LISA***α.

5. Tally: $\mathcal{V}$rf outputs $Attest$ and $Norep$ and returns to Wait.

Algorithm 5: $\mathcal{V}$rf pseudo-code in ***LISA***s.

```
$t_{ACK} \leftarrow t_{MAC} + 2t_t + t_s$;
$t_{Attest} \leftarrow n \cdot (t_{ACK} + t_a + t_{MAC} + t_t + t_s)$;
while $True$ do
    $wait()$;
    $\mathcal{I}nitID \leftarrow$ GETINITID();
    $\mathcal{C}urSeq \leftarrow$ GETSEQ();
    $\mathcal{A}tt_{req} \leftarrow$ "$req$"$||\mathcal{V}$rf$||\mathcal{C}urSeq||\mathcal{A}uth_{req}$;
    UNICAST($\mathcal{I}nitID, \mathcal{A}tt_{req}$);
    $Attest \leftarrow \emptyset$; $Children \leftarrow \emptyset$;
    $Norep \leftarrow \{all\mathcal{D}evID\}$;
    $T \leftarrow$ GETTIMER();
    while $T < t_{ACK}$ do
        $\mathcal{A}tt_{ack} \leftarrow$ RECEIVE();
        [$\mathcal{S}eq$, $\mathcal{D}evID$, $\mathcal{P}ar$] $\leftarrow$ DECOMPOSE($\mathcal{A}tt_{ack}$);
        if $\mathcal{S}eq = \mathcal{C}urSeq$ then
            $Children \leftarrow Children \cup \{\mathcal{D}evID\}$;
        end
    end
    while $T < t_{REP}$ do
        $\mathcal{A}tt_{rep} \leftarrow$ RECEIVE();
        [$\mathcal{S}nd$, $\mathcal{P}ar$, $\mathcal{S}eq$, $\mathcal{S}ndDesc$, $\mathcal{A}uth_{rep}$] $\leftarrow$ DECOMPOSE($\mathcal{A}tt_{rep}$);
        if $\mathcal{S}eq = \mathcal{C}urSeq \wedge \mathcal{A}uth_{rep} =$ MAC($K$, "$rep$"$||\mathcal{S}eq||\mathcal{D}evID||\mathcal{S}ndDesc$) then
            $Attest \leftarrow (Attest \cup \mathcal{S}ndDesc) \cup \{\mathcal{S}nd\}$;
            $Norep \leftarrow (Norep \setminus \mathcal{S}ndDesc) \setminus \{\mathcal{S}nd\}$;
            $Children \leftarrow Children \setminus \{\mathcal{S}nd\}$;
        end
        if $Children = \emptyset \vee |Attest| = n$ then
            BREAK();
        end
    end
    OUTPUT($Attest, Norep$);
end
```

Timeouts in *LISAs*

$\mathcal{P}\mathsf{rv}$ requires two timeouts: an acknowledgment timeout (t_{ACK}) and a report timeout (t_{REP}).

t_{ACK} is the amount of time that $\mathcal{P}\mathsf{rv}$ waits after having broadcast a request for children acknowledgments. It is set to the constant value of $t_{MAC} + 2t_t + t_s$, that is time for the broadcast to reach a neighbor (t_t), for the neighbor to verify the request (t_{MAC}), and then for the answer to be received by $\mathcal{P}\mathsf{rv}$ (another t_t), plus some slack t_s (global parameter). This gives all neighbors an opportunity to send $\mathcal{A}tt_{ack}$.

t_{REP} is the amount of time, after the children have been determined, that $\mathcal{P}\mathsf{rv}$ will wait for reports before performing its own attestation. It is set to the value $(n - \mathcal{D}epth)(t_{ACK} + t_a + t_{MAC} + t_t + t_s)$. This represents the time the descendants would take to answer back to $\mathcal{P}\mathsf{rv}$ in the absolute worst scenario. This scenario is when the descendants are in a line topology and each has only one child (except the last one). It is indeed the worst case because no work can be done in parallel. Each node in the line (except the leaf) will perform the following: forward a request to and wait for the answer from its (only) child (t_{ACK}), and then, after receiving the answer, verify its child's report (t_{MAC}), attest itself (t_a), and finally answer back to the parent (t_t) and some slack t_s for variability considerations. In this scenario, all of $\mathcal{P}\mathsf{rv}$'s descendants will take this time (except the leaf which takes slightly less). If $\mathcal{P}\mathsf{rv}$ has $\mathcal{D}epth$ ancestors, it has at most $(n-\mathcal{D}epth)$ descendants. Hence, time for attestation of descendants is bounded, even in the worst-case scenario, by: $(n - \mathcal{D}epth)(t_{ACK} + t_a + t_{MAC} + t_t + t_s)$. Note that the timeouts are needed to detect errors and DoS attacks. In most topologies, the delay in gathering reports will be much shorter. Finally, $\mathcal{D}epth$ is important: without it, if a node times out, its parent (and all ancestors) will also time out. Then, $\mathcal{V}\mathsf{rf}$ would have no idea about what happened. Instead, if a node times out, it sends what it has thus far to its parent, which does not time out itself.

Since $\mathcal{V}\mathsf{rf}$ can be viewed as the root of the spanning tree, t_{ACK} and t_{REP} are applicable to it. $\mathcal{V}\mathsf{rf}$'s $t_{REP} = n \cdot (t_{ACK} + t_a + t_{MAC} + t_t + t_s)$.

Connectivity in ***LISA****s*

Let t_0 denote the time that $\mathcal{P}\mathsf{rv}$ receives $\mathcal{A}uth_{req}$ from $\mathcal{P}ar$, t_1 denote the time that $\mathcal{P}ar$ receives $\mathcal{A}tt_{ack}$ from $\mathcal{P}\mathsf{rv}$ and t_2 denote the time that $\mathcal{P}ar$ receives $\mathcal{A}uth_{rep}$ from $\mathcal{P}\mathsf{rv}$. Then, we can formally state the connectivity assumption in ***LISA****s* as follows:

LISA*s* provides correct cRA result, i.e. no false-positive and false-negative, if a link between every $\mathcal{P}\mathsf{rv}$ and its $\mathcal{P}ar$ exists during their t_0, t_1 and t_2.

Note that this assumption is more relaxed than the one in ***LISA***α since each link has to appear during those three times while in ***LISA***α, $\mathcal{P}\mathsf{rv}$ and $\mathcal{P}ar$ have to be connected for transmitting $z + 1$ messages where z is a number of $\mathcal{P}\mathsf{rv}$'s descendants. The downside, however, is that when the assumption does not hold, $\mathcal{V}\mathsf{rf}$ will lose all $\mathcal{A}uth_{rep}$-s of $\mathcal{P}\mathsf{rv}$ and its descendants while in ***LISA***α, some of those $\mathcal{A}uth_{rep}$-s could still arrive to $\mathcal{V}\mathsf{rf}$.

QoSA of ***LISA****s*

As presented in Algorithm 4, ***LISA****s* offers L-QoSA. By changing information contained in the reports (line 48), QoSA can be amended to:

- *Binary:* Instead of $\mathcal{D}esc$, $\mathcal{A}tt_{rep}$ contains a single bit indicating whether all descendants have been successfully attested. This saves bandwidth over L-QoSA since reports are smaller (the MAC is also faster to compute). The obvious downside is that $\mathcal{V}\mathsf{rf}$ is only learns the result of cRA as a whole, and can not identify missing devices. One option is to use ***LISA****s* with B-QoSA until failure is encountered and then re-run ***LISA****s* with higher QoSA to identify devices that failed attestation.
- *Counter:* Using a counter allows $\mathcal{V}\mathsf{rf}$ to learn how many devices failed attestation. This

comes at a marginal increase is bandwidth consumption.

- *List Complement:* Instead of composing a list of attested descendants, each device can build a list of unattested ones. In a mostly healthy swarm, this is cheaper in terms of bandwidth than list QoSA.
- *Topology:* By representing the list of descendants as a tree instead of a set in the report, ***LISA****s* can provide topology information to $\mathcal{V}\mathsf{rf}$. Specifically, line 39 is replaced by: $\mathcal{D}esc \leftarrow \mathcal{D}esc \cup (\mathcal{S}nd : \mathcal{S}ndDesc)$. This recursively creates a subtree rooted at each node. The only drawback is a small increase in bandwidth usage.

Complexity of ***LISA****s*

Architectural Impact: LISA*s* does not require any additional secure hardware features, and, in coarse terms, adheres to SMART+. However, ROM needs to be expanded by around 200 bytes to support larger code. Also, ***LISA****s* has 5 write-protected values (while SMART+ has one): $\mathcal{S}eq$, $\mathcal{P}ar$, $\mathcal{C}$ and $\mathcal{D}esc$. To guard them, MPU needs to include at least as many memory access control rules. Each of the first two is a 4-byte integer, while $\mathcal{C}$ is 32 bytes, while the size of $\mathcal{D}esc$ is proportional to n. In total, $\mathcal{P}\mathsf{rv}$ needs $40{+}4n$ bytes of write-protected memory, which is $O(n)$. Protecting a fixed-size value is clearly easier than a variable-size one. Nonetheless, we illustrates a simple mechanism that accommodates variable-size data with implicit write-protection with minimal (constant) added space complexity for write-protected memory below:

Let x be a variable-sized data that needs to be write-protected. Let h_x be a fixed-size memory location that stores $H(x)$. Instead of enforcing access rules for x, the MPU ensures that only h_x is write-protected. Whenever x is modified to x', MPU stores $H(x')$ at h_x. Whenever x (as a whole or any part thereof) needs to be read, MPU first checks whether $h_x = H(x)$. This does not prevent malware from modifying x. However, any unauthorized change is detected upon the next read, which is sufficient for our purposes.

We note that SEDA already implicitly requires write-protected variable-sized data even though [7] does not discuss how this can be achieved in practice.

Software Complexity: ***LISA***s's software is much more complex than that of SMART+ or ***LISA***α. Compared to ***LISA***α, ***LISA***s has three extra states, needed to: (1) determine timeouts, (2) establish parent-child relationship, and (3) handle report aggregation. For that, $\mathcal{P}\mathsf{rv}$ needs to store additional variables(two of which are arrays): $\mathcal{D}epth$, $\mathcal{P}ar$, $\mathcal{D}esc$ and $\mathcal{C}hildren$, in each attestation session. This makes ***LISA***s a stateful protocol. In terms of LoC-s, ***LISA***s is approximately 22% and 19% over SMART+ and ***LISA***α, respectively.

Bandwidth Usage: Suppose $\mathcal{P}\mathsf{rv}$ has q children, z descendants and w neighbors. Compared to ***LISA***α, $\mathcal{A}tt_{req}$ includes one extra field: $\mathcal{D}epth$– 4 bytes. Thus, the size of $\mathcal{A}tt_{req}$ in ***LISA***s is 47 bytes. $\mathcal{A}tt_{rep}$ does not include $H(\mathcal{M}em)$ and $\mathcal{P}ar$. However, it contains additional variable-size data, $\mathcal{D}esc$, which can be as long as $4z$. Thus, the size of $\mathcal{A}tt_{rep}$ is $47 + 4z$ bytes. Finally, $\mathcal{A}tt_{ack}$ size is 15 bytes: 3 – message tag, 4 – $\mathcal{S}eq$, 4 – $\mathcal{D}evID$ field and 4 – $\mathcal{P}ar$.

In each session, $\mathcal{P}\mathsf{rv}$ broadcasts one $\mathcal{A}tt_{req}$ to its neighbors, plus unicasts one $\mathcal{A}tt_{rep}$ and one $\mathcal{A}tt_{ack}$ to $\mathcal{P}ar$. Thus, the overall transmission cost for $\mathcal{P}\mathsf{rv}$ is $47 + 47 + 4z + 15 = 109 + 4z$. In the same session, $\mathcal{P}\mathsf{rv}$ receives up to w $\mathcal{A}tt_{req}$-s, exactly q $\mathcal{A}tt_{rep}$-s and q $\mathcal{A}tt_{ack}$-s. Hence, in the worst case, $\mathcal{P}\mathsf{rv}$ receives (in bytes):

$$47w + \sum_{i=1}^{q}(47 + 4z_i) + 15q = 47w + 47q + 4(z - q) + 15q = 47w + 58q + 4z$$

where z_i is the number of descendants of i^{th} child of $\mathcal{P}\mathsf{rv}$.

Overall, ***LISA***s reduces the number of messages, compared to ***LISA***α. Each $\mathcal{P}\mathsf{rv}$ transmits a fixed number of messages while, in ***LISA***α, this depends on the number of descendants and neighbors.

8.4 Security Analysis

We now describe possible attacks and then (informally) assess security of ***LISA***α and ***LISA***s.

8.4.1 Attack Vectors

Recall that our adversarial model only considers *remote and local adversaries*, while physical attacks are out of scope. An adversary $\mathcal{A}dv$ can remotely modify the software and/or the state of any device. It also has full control of all communication channels, i.e., can eavesdrop on, inject, delete, delay or modify any messages between devices, as well as between any device and $\mathcal{V}\mathsf{rf}$. In the context of ***LISA***, the following attacks are possible:

1. *Report Forgery:* Forging a $\mathcal{A}tt_{rep}$ would allow a device to evade detection of malware, or allow $\mathcal{A}dv$ to impersonate a device.
2. *Request Forgery:* Forging a $\mathcal{A}tt_{req}$ would trigger unnecessary cRA and result in denial-of-service (DoS) for the entire swarm.
3. *Application Layer DoS:* $\mathcal{A}dv$ can launch a DoS attack abusing the attestation protocol itself. This type of attack can vary, depending on the protocol version. One example is flooding the swarm with fake $\mathcal{A}tt_{rep}$-s.
4. *DoS on Network Layer and Lower Layers:* $\mathcal{A}dv$ can launch DoS attacks that target network, link and physical layers of devices. This includes radio jamming, random packet flooding, packet dropping, etc. We do not consider such attacks since they are not specific to cRA.

8.4.2 Security of $\boldsymbol{LISA\alpha}$

Report Forgery: Recall that $\mathcal{A}tt_{rep}$ in $\boldsymbol{LISA\alpha}$ is $[\text{“}rep\text{”}, \mathcal{D}evID, \mathcal{P}ar, \mathcal{S}eq, H(\mathcal{M}em), \mathcal{A}uth_{rep}]$ where $\mathcal{A}uth_{rep} = \mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{D}evID||\mathcal{S}eq||H(\mathcal{M}em))$. If $\mathcal{A}dv$ produces an authentic $\mathcal{A}tt_{rep}$ for some device then one of the following must hold:

- $\mathcal{A}dv$ forges $\mathcal{A}uth_{rep}$ without knowing K: this requires $\mathcal{A}dv$ to succeed in a MAC forgery, which is infeasible, with overwhelming probability, given a secure MAC function.
- $\mathcal{A}dv$ knows K and constructs its own $\mathcal{A}uth_{rep}$: this is not possible, since only $\boldsymbol{AttCode}$ can read K, $\boldsymbol{AttCode}$ leaks no information about K beyond $\mathcal{A}uth_{rep}$, and $\mathcal{A}dv$ can not tamper with hardware.
- $\mathcal{A}dv$ modifies a compromised device's $\mathcal{D}evID$ which results in producing $\mathcal{A}uth_{rep}$ for another $\mathcal{D}evID$: since $\mathcal{D}evID$ “lives” in ROM, it can not be modified.

We note that replay attacks are trivially detected by $\mathcal{V}\mathsf{rf}$ since each $\mathcal{A}tt_{req}$ includes a unique monotonically increasing $\mathcal{S}eq$, which is authenticated by every $\mathcal{P}\mathsf{rv}$ and included in $\mathcal{A}tt_{rep}$.

Request Forgery: Recall that $\mathcal{A}tt_{req}$ in $\boldsymbol{LISA\alpha}$ is $[\text{“}req\text{”}, \mathcal{S}nd, \mathcal{S}eq, \mathcal{A}uth_{req}]$ where $\mathcal{A}uth_{req} = \mathsf{MAC}(K, \text{“}req\text{”}||\mathcal{S}eq)$. An $\mathcal{A}dv$ that fakes an $\mathcal{A}tt_{req}$ must either create a forged $\mathcal{A}uth_{req}$ without K or somehow know K. Similar to report forgery above, neither case is possible due to our assumptions about the MAC function and inaccessibility of K.

Application Layer DoS: An $\mathcal{A}dv$ flooding the swarm with fake $\mathcal{A}tt_{rep}$-s and/or $\mathcal{A}tt_{req}$-s can result in an effective attack if it triggers a lot of computation on, and/or communication between, devices. Fake $\mathcal{A}tt_{req}$ flooding to a device is not very effective since it causes DoS for only that device which authenticates an $\mathcal{A}tt_{req}$ before doing further work and forwarding it. On the other hand, a fake $\mathcal{A}tt_{rep}$ sent to a single device can result in several devices forwarding garbage. This is because a device forwards a report to its parent (and further up the spanning tree) without any authentication. We consider this attack not to be severe because it does not trigger any other computation (only communication).

8.4.3 Security Analysis of *LISA*s

Report Forgery: Analysis of this attack in ***LISA***s is similar to that in ***LISA***α. $\mathcal{A}tt_{rep}$ in ***LISA***s is $[\text{“}rep\text{”}, \mathcal{D}evID, \mathcal{C}urSeq, \mathcal{D}esc, \mathcal{A}uth_{rep}]$, where $\mathcal{A}uth_{rep} = \mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{D}evID ||\mathcal{C}urSeq||\mathcal{D}esc)$. If $\mathcal{A}dv$ successfully forges a $\mathcal{A}tt_{rep}$ for one of the swarm devices such that $\mathcal{V}\mathsf{rf}$ accepts it, then one of the following must have occurred: (1) $\mathcal{A}dv$ forged $\mathcal{A}uth_{rep}$ violating security of the underlying MAC; (2) $\mathcal{A}dv$ learned K which is in ROM and only accessible from ***AttCode*** which is leak-proof; or (3) $\mathcal{A}dv$ was able to modify variables that “live” in write-protected memory (i.e., $\mathcal{C}urSeq$, $\mathcal{D}evID$, $\mathcal{P}ar$, $\mathcal{D}esc$ and $\mathcal{C}hildren$). This is also not possible due to the guaranteed write-protection from MPU access rules and ROM.

Request Forgery: $\mathcal{A}tt_{req}$ in ***LISA***s is similar to $\mathcal{A}tt_{req}$ in ***LISA***α except the additional field – $\mathcal{D}epth$. This field is, however, not utilized when checking integrity of $\mathcal{A}tt_{req}$. Thus, the analysis of this attack is similar to that in the ***LISA***α case above.

Application-Layer DoS: Fake request flooding in ***LISA***s has the same effect as that in ***LISA***α since the request of both protocols has similar format and is handled similarly. Fake report flooding, nonetheless, does not result in significant communication overhead since a device in ***LISA***s verifies all reports before aggregating them. In addition, ***LISA***s involves one additional type of message: $\mathcal{A}tt_{ack}$. Recall that $\mathcal{P}\mathsf{rv}$ in ***LISA***s constructs $\mathcal{C}hildren$ based on $\mathcal{A}tt_{ack}$-s and then waits for reports until $\mathcal{C}hildren$ is empty or a timeout t_{REP} occurs. A fake $\mathcal{A}tt_{ack}$ causes devices to wait longer than necessary. However, such waiting time is still bounded by t_{REP} and thus in the worst case this attack will result in timeout of all of its ancestor devices. This attack is not severe since it does not incur extra computation on any devices, and produces effects similar to DoS attacks on lower layers. Fake $\mathcal{A}tt_{ack}$ flooding is also possible in ***LISA***s, though it results in DoS for targeted devices and not the entire swarm.

8.5 Experimental Assessment

We implemented ***LISA**α* and ***LISA**s* in Python, and assessed their performance by emulating device swarms using the open-source Common Open Research Emulator (CORE) [3].

8.5.1 Experimental Setup and Parameters

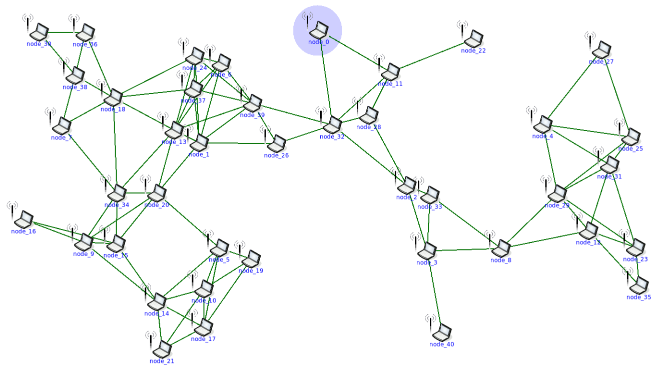


Figure 8.4: Example scenario generated in CORE (40 nodes). $\mathcal{V}\mathsf{rf}$ is highlighted.

CORE Emulator: CORE is a framework for emulating networks. It allows defining network topologies with lightweight virtual machines (VMs). CORE contains Python modules for scripting network emulations at different layers of the TCP/IP stack and allows defining mobility scenario for nodes, configuring routing and switching, network traffic and applications running inside emulated VMs. One key advantage of CORE over other simulation frameworks, such as ns2 or ns3, is that it utilizes the actual network stack in Linux and instantiates each emulated node using a lightweight VM with its own file system running its own processes. Using the actual networking stack results in performance estimates very close to reality, since it does not abstract away any implementation details or issues at the data link, network and transport layers.

Experimental Setup & Scenarios: We generated several CORE scenarios with n nodes each. In every scenario, the positions of the n nodes are chosen uniformly at random in an area of $1,500\times$ 800 units, e.g., meters. A pair of nodes is connected if the distance between them is smaller then a threshold of 200 units, corresponding roughly to the coverage range of 802.11/WiFi. If the resulting network is not connected, the above process is repeated until a connected network is generated. $\mathcal{V}\mathsf{rf}$ is also randomly positioned within the area, and connected to the generated network. Figure 8.4 in shows a sample configuration.

The link-layer medium access control protocol running between neighboring nodes is 802.11. Network layer (IP) routing tables are automatically populated via the Optimized Link State Routing (OLSR) protocol, an IP-based routing protocol optimized for MANETs. OLSR uses proactive link-state routing with "Hello" and "Topology Control" messages that discover and disseminate link state information throughout the network. Each node uses topology information to compute next-hop destinations for all other nodes using the shortest path algorithm. Each node then runs our cRA protocol, though instead of actually performing cryptographic operations, we insert delays (specified below) corresponding to time to perform such operations on low-end devices. At the beginning of each experiment, $\mathcal{V}\mathsf{rf}$ broadcasts a new $\mathcal{A}tt_{req}$ that is propagated throughout the network according to $\boldsymbol{LISA}\alpha$ or $\boldsymbol{LISA}s$.

Timings of Cryptographic Operations: Delays used to mimic cryptographic operations on low-end devices are as follows:

- $\mathcal{V}\mathsf{rf}$ signature computation/verification: $0.0001s$
- Node signature computation/verification: $0.001s$
- Node hash speed (for attestation): $0.0429s$/MB

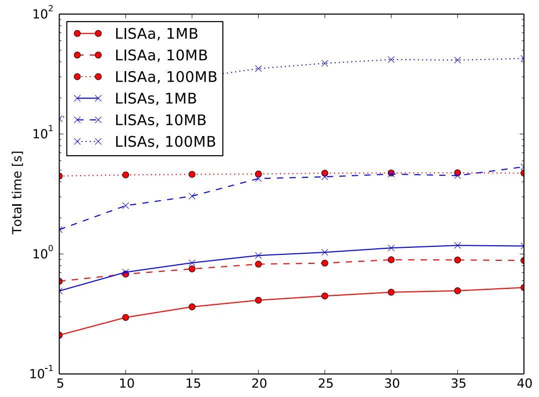
The scheme used to sign messages can be implemented by a MAC or a public key signature scheme such as ECDSA (see Section 8.6). These timings are based on using ECDSA-160 and SHA-256. Using a MAC would reduce the time for small memory sizes. However, as discussed in Sections 8.5.2 and 8.6, computation is generally dominated by hashing. Numbers for $\mathcal{V}\mathsf{rf}$ are derived from a typical laptop, and those for $\mathcal{P}\mathsf{rv}$ nodes come from a Raspberry Pi-2.

8.5.2 Experimental Results

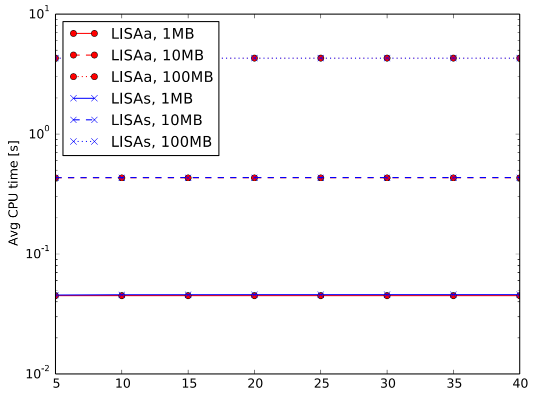
In each experiment, we measured: (a) total time to perform cRA: from $\mathcal{V}\mathsf{rf}$ sending $\mathcal{A}tt_{req}$, until $\mathcal{V}\mathsf{rf}$ finishes verification of the last $\mathcal{A}tt_{rep}$; (b) average CPU time for $\mathcal{P}\mathsf{rv}$ to performing attestation; and (c) average number of bytes transmitted per $\mathcal{P}\mathsf{rv}$. Figure 8.5 shows the results for both protocols with various amounts of attested memory and different swarm sizes. Each data point is obtained as an average over 30 randomly generated scenarios for that setup.

Total time (Figure 8.5a) varies significantly between $\boldsymbol{LISA}\alpha$ and $\boldsymbol{LISA}s$. This is because in $\boldsymbol{LISA}s$ nodes spend a lot of time waiting for external input, without computing anything. In these results, the factor varies between 2 (for 1MB) to 8 (for 100MB). This time is also heavily influenced by the size of the attested memory, as shown in Figure 8.5b. Finally, total attestation time depends (roughly logarithmically) on n, since nodes are explored in a tree fashion. Although random, the tree is expected to be relatively well-balanced (see Figure 8.4).

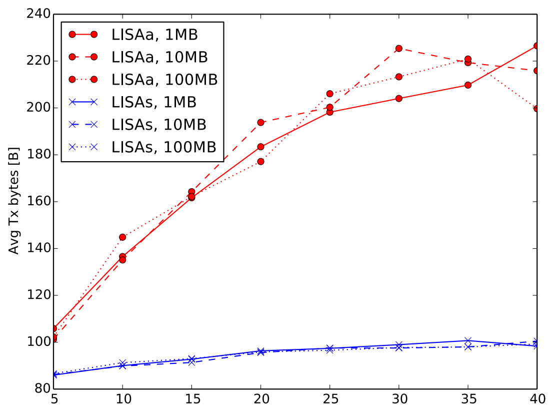
Average CPU time (Figure 8.5b) for $\mathcal{P}\mathsf{rv}$ is roughly equivalent in both protocols. This might seem counterintuitive, since in $\boldsymbol{LISA}s$ nodes verify $\mathcal{A}tt_{rep}$-s of their children. However, verification is much cheaper than attestation, in particular, if attested memory size is large. This is discussed in Section 8.6. The number of devices (n) also has little effect on computation costs. On the other hand, the amount of attested memory has a strong impact on $\mathcal{P}\mathsf{rv}$'s CPU usage. This shows that both protocols impose negligible extra overhead (over single-prover attestation) in terms of CPU usage.



(a) Total time [s] for cRA in ***LISA***α and ***LISA***s, for different memory sizes, as a function of n (log y-scale).



(b) Average CPU time [s] per device for ***LISA***α and ***LISA***s, for different memory sizes, as a function of n (log y-scale).



(c) Average # bytes transmitted per device for ***LISA***α and ***LISA***s, for different memory sizes, as a function of n (linear y-scale).

Figure 8.5: Experimental Results for ***LISA***α and ***LISA***s

Bandwidth usage (Figure 8.5c) is, as expected, higher in $\boldsymbol{LISA}\alpha$ than in $\boldsymbol{LISA}s$. The exact difference factor depends on n, ranging from negligible (5 nodes) to 3 (40 nodes). This only represents payloads size. Nodes in $\boldsymbol{LISA}\alpha$ also send more packets[7], compared to only 3 in $\boldsymbol{LISA}s$: $\mathcal{A}tt_{req}$, $\mathcal{A}tt_{ack}$, and $\mathcal{A}tt_{rep}$. Bandwidth usage is also roughly linear in terms of n. The size of the memory does not affect bandwidth usage, since data transmitted by nodes is independent to it.

8.6 Cryptographic Choices

As described above, both $\boldsymbol{LISA}\alpha$ and $\boldsymbol{LISA}s$ assume that symmetric cryptography is used for constructing the MAC primitive, i.e., a keyed cryptographic hash function [9] or a CBC-based MAC [41]. Key management is trivial: a single master key shared between $\mathcal{V}\mathsf{rf}$ and all devices is used for computing and verifying all attestation reports. However, under some conditions, it might be desirable or even preferable to apply less naïve key management techniques and/or take advantage of public-key cryptography.

Physical Attacks: As stated earlier, $\boldsymbol{LISA}\alpha$ and $\boldsymbol{LISA}s$ consider physical attacks to be out of scope, following most prior literature on this topic. Thus, SMART+ architecture, coupled with a single shared symmetric master key, is sufficient for attesting the entire swarm in the presence of *Remote* and *Local* adversaries, as defined in Section 8.2.2. However, in the presence of physical adversaries, neither scheme is secure. A physical attack on a single device exposes the master key, which allows the adversary to impersonate all other devices as well as $\mathcal{V}\mathsf{rf}$.

Device-Specific Keying: One natural mitigation approach is to impose a unique key that each device shares with $\mathcal{V}\mathsf{rf}$. That way, the adversary learns only one key upon compromising a

[7] The number of packets sent by both protocols, not depicted here, follows a behavior very similar to Figure 8.5c.

single device. Although this approach would work well with ***LISA***α, it requires changes to ***LISA***s to support key establishment among neighboring devices; this would likely entail the use of public key cryptography, e.g., using the Diffie-Hellman key establishment protocol.

$\mathcal{A}tt_{req}$ Authentication: Device-specific symmetric keying also does not address the issue of $\mathcal{V}\mathsf{rf}$ impersonation. To allow devices to authenticate each $\mathcal{A}tt_{req}$ individually, $\mathcal{V}\mathsf{rf}$ would need to compute n distinct $\mathcal{A}uth_{req}$ tags, one for each device. This might incur significant computational and bandwidth overheads, if n is large. For small swarms, the tradeoff could be acceptable. Regardless of whether a single master key or device-specific keys are used, one simple step towards preventing $\mathcal{V}\mathsf{rf}$ impersonation and consequent DoS attacks is to impose a public key on $\mathcal{V}\mathsf{rf}$ only. In other words, $\mathcal{V}\mathsf{rf}$ would sign each $\mathcal{A}tt_{req}$, thus changing the format of $\mathcal{A}uth_{req}$ to: $SIGN(SK_{\mathcal{V}\mathsf{rf}}, \text{"}req\text{"}||\mathcal{S}eq)$ where $SK_{\mathcal{V}\mathsf{rf}}$ is $\mathcal{V}\mathsf{rf}$'s private key and $PK_{\mathcal{V}\mathsf{rf}}$ is its public counterpart, assumed to be known to all devices. One obvious downside of this simple method is the extra computational overhead of verifying $\mathcal{A}uth_{req}$. We note that the combination of: (1) public key-based $\mathcal{A}tt_{req}$ authentication, and (2) per device symmetric keys, is both appropriate and efficient for ***LISA***α, which does not require devices to authenticate each other's $\mathcal{A}tt_{rep}$-s. It makes less sense for ***LISA***s, due to the need for a means to authenticate one's neighbors' $\mathcal{A}tt_{rep}$-s.

Public Keys for All: Predictably, we now consider imposing a unique public/private key-pair for each device. Admittedly, this only mitigates the effects of physical attacks and clearly does not prevent them. However, a successful physical attack on a single device yields knowledge of that device's secret key and does not lead to impersonation, or easier compromise, of other devices. For ***LISA***α, there is almost no difference between the full-blown public key approach and device-specific symmetric keying, as long as either is coupled with public key-based $\mathcal{A}tt_{req}$ authentication. The only issue arises if $\mathcal{V}\mathsf{rf}$ is not fully trusted; in that case, the former is preferable since $\mathcal{V}\mathsf{rf}$ would not be able to create fake $\mathcal{A}tt_{rep}$-s. For ***LISA***s, using device-specific public keys is conceptually simpler as there would be no need to establish pairwise keys between neighbors.

Attested Memory Size: An orthogonal (non-security) issue influencing cryptographic choices is the size of attested memory. Considering relatively weak low-end embedded devices, the cost of computing a symmetric MAC (dominated by computing a hash) over a large segment of memory might exceed that of computing a single public key signature. In that case, it makes sense to employ a digital signature in both ***LISA***α and ***LISA***s. To illustrate this point, Figure 8.6 compares performance of SHA-256 with several signature algorithms on Raspberry Pi-2. When attested memory size reaches 45KB, the run-time of Elliptic Curve Digital Signature Algorithm (ECDSA) with a 256-bit public key catches up to that of SHA-256. Hence, at least with Raspberry Pi-2, it makes sense to switch to ECDSA-256 for memory sizes exceeding 4.5MB – at that point, ECDSA-256 consumes less than 1% of total attestation runtime.

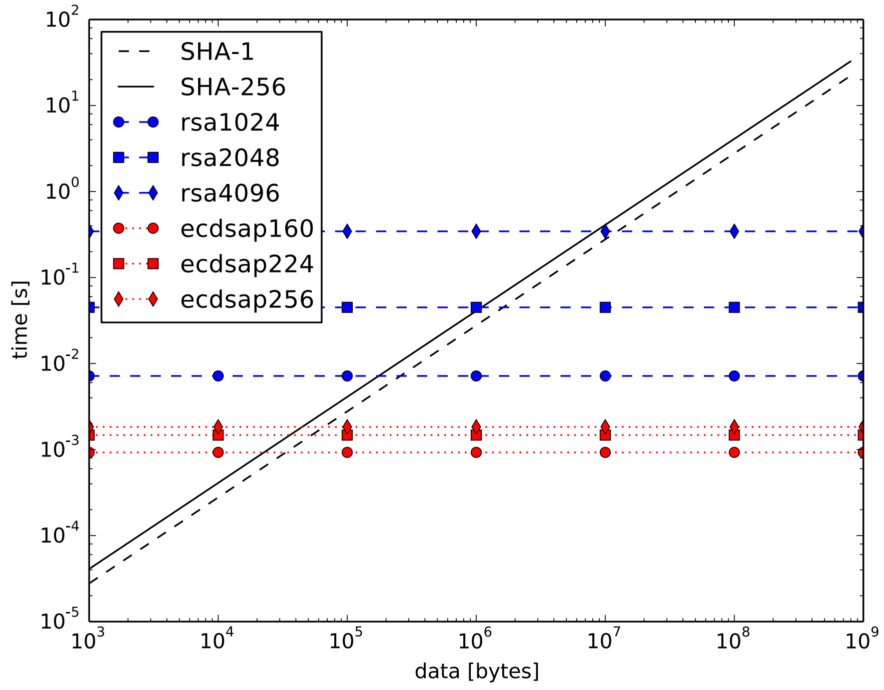


Figure 8.6: Performance Comparison: Hash & Signature on Raspberry Pi-2@900MHz [72].

8.7 Applicability in Security-Critical Applications

The current versions of ***LISA*** keep $\mathcal{P}\mathsf{rv}$-s busy for a non-negligible amount of time during cRA (See our evaluation results in Section 8.5.2 for more details). Naturally, this makes them unsuitable for time-/safety-critical swarm devices that operate under strict time constraints. One direction to resolve this issue is to minimize the real-time requirement in ***LISA*** by adopting the periodic self-measurement technique, which is thoroughly discussed in Chapter 7.

Consider a variant of ***LISA***α in which all $\mathcal{P}\mathsf{rv}$-s deploy ERASMUS-based self-measurements instead of performing on-demand $\mathcal{R}\mathsf{A}$ based on a $\mathcal{V}\mathsf{rf}$ request. In this variant, each $\mathcal{P}\mathsf{rv}$ periodically records its software state, and simply transmits the latest recording to its parent during cRA. Doing so eliminate the need for $\mathcal{P}\mathsf{rv}$-s to perform on-demand $\mathcal{R}\mathsf{A}$, which represents the main bottleneck in ***LISA***α. As no on-demand $\mathcal{R}\mathsf{A}$ is required, this variant is not exposed to computational DoS attacks, entailing that authentication of $\mathcal{V}\mathsf{rf}$ requests is also no longer necessary. As a result, this variant of ***LISA***α imposes no time-consuming computation on $\mathcal{P}\mathsf{rv}$-s and also allows more flexible scheduling of $\mathcal{R}\mathsf{A}$, making it suitable for a swarm of safety-critical devices. The same technique is less effective on ***LISA***s, however. This is because, even coupling with the self-measurement technique, each $\mathcal{P}\mathsf{rv}$ in ***LISA***s still needs to verify its children's reports in order to prevent them from lying, and this process can be potentially time-consuming (even though much less time-consuming than on-demand $\mathcal{R}\mathsf{A}$).

Chapter 9

Conclusion & Future Work

This dissertation proposed several mitigation mechanisms to remedy the conflict between requirements of secure remote attestation and those of safety-critical operations.

We started off in Chapter 4 by discussing HYDRA – the underlying $\mathcal{R}\mathsf{A}$ architecture in our mitigation mechanisms. HYDRA follows the same design principle as hybrid $\mathcal{R}\mathsf{A}$, which aims to minimize hardware features on $\mathcal{P}\mathsf{rv}$. For software components, we leveraged the seL4 formally verified microkernel to enforce memory isolation and access control to resources in HYDRA. We demonstrated how to derive seL4's access control configurations from security properties of hybrid $\mathcal{R}\mathsf{A}$. Contrary to previous hybrid $\mathcal{R}\mathsf{A}$ techniques, HYDRA does not require any changes to the underlying processor; the only hardware support needed by HYDRA is a hardware-enforced secure boot feature, which is readily available on commercially available development boards and processors. This makes HYDRA deployable even on the existing off-the-shelf IoT devices. Admittedly, despite building on top of the formally verified kernel, the user-space components of HYDRA are not verified. One promising future direction is to formally verify such components with respect to formal security properties and functional correctness.

Using HYDRA as a building block, we presented the first mitigation mechanism, called *SMARM*, in Chapter 5. *SMARM* aims to reconcile secure $\mathcal{R}\mathsf{A}$ with safety-critical applications by carefully relaxing the atomicity requirement in $\mathcal{R}\mathsf{A}$. In particular, *SMARM* permits the measurement process to be interruptible while mitigating self-relocating malware by measuring $\mathcal{P}\mathsf{rv}$'s memory in a random and secret order. This order is established randomly when the measurement starts, and then stored in secure memory. *SMARM* requires the memory measurement to be performed multiple times in order to ensure a negligible probability of malware evasion. Thus, this technique imposes additional run-time overhead on $\mathcal{P}\mathsf{rv}$. Nonetheless, we showed that $\mathcal{R}\mathsf{A}$ based on *SMARM* minimizes practical impact on the availability of $\mathcal{P}\mathsf{rv}$ for its main operation, which is the primary requirement in safety-critical applications.

Chapter 6 introduced the notion of temporal consistency with a focus on the $\mathcal{R}\mathsf{A}$ application. We argued that, in practice, inputs to any cryptographic integrity-ensuring functions can change during computation, and thus the result from such computation may be *inconsistent*, i.e., it may reflect a state of inputs that never existed in their entirety at any given time. We presented various memory locking mechanisms to ensure temporal consistency in $\mathcal{R}\mathsf{A}$. They offer tradeoffs between consistency guarantees, performance overhead, and impact on memory availability. We implemented the proposed mechanisms on two commodity platforms using HYDRA as the underlying $\mathcal{R}\mathsf{A}$ architecture. Experimental results showed that our mechanisms can be achieved with less than 10% overhead on both platforms, while providing much better availability for safety-critical applications.

Chapter 7 presented ERASMUS as an alternative to current on-demand $\mathcal{R}\mathsf{A}$ techniques for IoT devices. It is based on scheduled self-measurements, which allows periodically and autonomously recording $\mathcal{P}\mathsf{rv}$'s software state without relying on $\mathcal{V}\mathsf{rf}$'s interaction. Using this technique, $\mathcal{P}\mathsf{rv}$ has control over the measurement schedule and thus can avoid running the measurement process at the same time as safety-critical processes. It also provides detection of mobile malware, which is not possible with on-demand $\mathcal{R}\mathsf{A}$ techniques. Its other major advantage is that it requires no cryptographic computation as part of $\mathcal{V}\mathsf{rf}$'s interaction. This makes ERASMUS more resilient to DoS attacks based on $\mathcal{V}\mathsf{rf}$ impersonation. We also discussed the possibility of using on-demand $\mathcal{R}\mathsf{A}$ as part of ERASMUS collection phase to obtain maximal freshness.

Chapter 8 brought collective remote attestation (cRA) closer to reality by designing two simple and practical protocols: ***LISA****α* and ***LISA****s*. To analyze and compare across protocols, we introduced a new metric: Quality of Swarm Attestation (QoSA) which captures the information offered by a specific cRA protocol. We showed that ***LISA****s* can achieve more variety of QoSA-s. Meanwhile, the design of ***LISA****α* is simpler, resulting in a smaller size of secure memory storage and less complex software. We evaluated both ***LISA****s* and ***LISA****α* using the open-source CORE emulator. Evaluation results showed feasibility and practicality of our proposed protocols; both ***LISA****s* and ***LISA****α* take less than 1 minute to attest a swarm of up to 40 devices. We also described variants of ***LISA***-s that replace on-demand $\mathcal{R}\mathsf{A}$ by the self-measurement technique, making them good candidates for cRA of safety-critical devices.

Future Research Directions. All of our proposed mitigation mechanisms build on top of $\mathcal{R}\mathsf{A}$ techniques that compute an attestation report based on a memory measurement (e.g., keyed hash of $\mathcal{P}\mathsf{rv}$'s memory). Recall from Chapter 2 that such techniques only give $\mathcal{V}\mathsf{rf}$ an assurance about $\mathcal{P}\mathsf{rv}$'s software integrity during the measurement time; it does not guarantee that the measured memory will run properly. Various control-flow based $\mathcal{R}\mathsf{A}$ techniques [1, 25, 24] have been recently proposed in order to address this limitation. One interesting future research direction is to adapt our mitigation mechanisms to support such control-flow $\mathcal{R}\mathsf{A}$ techniques. This can especially be challenging since control-flow $\mathcal{R}\mathsf{A}$ techniques are known to be prone to the problem of execution-path (or state) explosion [1]; allowing interrupts to occur during control-flow $\mathcal{R}\mathsf{A}$ could further complicate this problem.

Next, security arguments of our proposed mechanisms are informally stated. The natural extension is to formally prove our design and implementation with respect to formal security guarantees. One way to achieve this is to follow the same verification methodology as the work in [65] (for single $\mathcal{P}\mathsf{rv}$ settings) or [64] (for multiple $\mathcal{P}\mathsf{rv}$-s settings), which would require the following: (1) formalize an end-to-end notion of secure $\mathcal{R}\mathsf{A}$ in the context of safety-critical settings, (2) use a formal verification framework (e.g., theorem prover or model checking) to prove that the protocol design of our proposed mechanism satisfies the end-to-end notion, and (3) prove that our implementation corresponds to the proven protocol design. The final direction for future work that we intend to pursue is trial deployment of our proposed mechanisms in real-world safety-critical settings. Doing so will demonstrate stronger practicality of our proposed mechanisms.

Bibliography

[1] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-flat: control-flow attestation for embedded systems software. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[2] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A.-R. Sadeghi, and G. Tsudik. Invited: Things, trouble, trust: on building trust in IoT systems. In *ACM/IEEE Design Automation Conference (DAC)*, 2016.

[3] J. Ahrenholz. Comparison of CORE network emulation platforms. In *IEEE Military Communications Conference (MILCOM)*, 2010.

[4] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security Symposium*, 2017.

[5] Apple Computer, Inc. LibOrange. `https://github.com/unixpickle/LibOrange/blob/master/LibOrange/hmac-sha256.c`, 2006.

[6] ARM Limited. ARM security technology - building a secure system using trustzone technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2009.

[7] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann. SEDA: Scalable embedded device attestation. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[8] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), 2018.

[9] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *IACR Crypto*, 1996.

[10] Boundary Devices. BD-SL-I.MX6. `https://boundarydevices.com/product/sabre-lite-imx6-sbc/`, 2017.

[11] F. Brasser, P. Koeberl, B. E. Mahjoub, A.-R. Sadeghi, and C. Wachsmann. TyTAN: tiny trust anchor for tiny devices. In *Design Automation Conference (DAC)*, 2015.

[12] F. Brasser, A.-R. Sadeghi, and G. Tsudik. Remote attestation for low-end embedded devices: the prover's perspective. In *ACM/IEEE Design Automation Conference (DAC)*, 2016.

[13] S. Bratus, N. DCunha, E. Sparks, and S. W. Smith. Toctou, traps, and trusted computing. In *International Conference on Trusted Computing*. Springer, 2008.

[14] J. Camhi. BI Intelligence projects 34 billion devices will be connected by 2020. `http://www.businessinsider.com/bi-intelligence-34-billion-connected-devices-2020-2015-11`, 2015.

[15] X. Carpent, K. Eldefrawy, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Invited: Reconciling remote attestation and safety-critical operation on simple iot devices. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018.

[16] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik. Lightweigh swarm attestation: a tale of two lisa-s. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[17] X. Carpent, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. Temporal consistency of integrity-ensuring computations and applications to embedded systems security. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[18] X. Carpent, N. Rattanavipanon, and G. 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*, 2018.

[19] X. Carpent, N. Rattanavipanon, and G. Tsudik. Remote attestation via self-measurement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24(1):11, 2018.

[20] X. Carpent, G. Tsudik, and N. Rattanavipanon. Erasmus: Efficient remote attestation via self-measurement for unattended settings. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[21] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM conference on Computer and Communications Security*, 2009.

[22] Y.-G. Choi, J. Kang, and D. Nyang. Proactive code verification protocol in wireless sensor network. In *International Conference on Computational Science and Its Applications*, 2007.

[23] V. Costan and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086, 2016.

[24] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi. Litehax: Lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[25] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Annual Design Automation Conference (DAC)*, 2017.

[26] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 1979.

[27] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[28] D. Eastlake 3rd and P. Jones. US secure hash algorithm 1 (sha1). Technical report, 2001.

[29] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. Fusing hybrid remote attestation with a formally verified microkernel: Lessons learned. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2017.

[30] K. Eldefrawy, N. Rattanavipanon, and G. 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.

[31] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[32] R. A. Fisher, F. Yates, et al. Statistical tables for biological, agricultural and medical research. *Statistical tables for biological, agricultural and medical research.*, 1949.

[33] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe (DATE)*, 2014.

[34] Freescale Semiconductor, Inc. i.MX 6 Linux High Assurance Boot (HAB) User's Guide. Technical report, 2013.

[35] O. Girard. OpenMSP430. `https://opencores.org/project/openmsp430`, 2009.

[36] V. D. Gligor and S. L. M. Woo. Establishing software root of trust unconditionally. In *Network and Distributed System Security Symposium*, 2019.

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

[38] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni. DARPA: Device attestation resilient to physical attacks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, New York, NY, USA, 2016. ACM.

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

[40] ISO/IEC. 19772:2009, Information technology – Security techniques – Authenticated encryption, 2009.

[41] Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher. Standard, ISO, 1999.

[42] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*, 1(1):36–63, 2001.

[43] A. Kandhalu, K. Lakshmanan, and R. R. Rajkumar. U-connect: a low-latency energy-efficient asynchronous neighbor discovery protocol. In *ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN)*, 2010.

[44] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):2, 2014.

[45] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *ACM Symposium on Operating systems principles (SOSP)*, 2009.

[46] D. E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

[47] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *European Conference on Computer Systems*, 2014.

[48] F. Kohnhäuser, N. Büscher, and S. Katzenbeisser. A practical attestation protocol for autonomous embedded systems. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[49] M. Kohvakka, J. Suhonen, M. Kuorilehto, V. Kaseva, M. Hännikäinen, and T. D. Hämäläinen. Energy-efficient neighbor discovery protocol for mobile wireless sensor networks. *Ad Hoc Networks*, 7(1):24–41, 2009.

[50] H. Krawczyk, R. Canetti, and M. Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[51] laginimaineb. Extracting qualcomm's keymaster keys! `https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html`, 2016.

[52] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *ACM Conference on Computer and Communications Security*, 2011.

[53] Y. Lindell and J. Katz. *Introduction to modern cryptography*, chapter 3, pages 68–72. Chapman and Hall/CRC, 2014.

[54] R. J. Lipton, R. Ostrovsky, and V. Zikas. Provably secure virus detection: Using the observer effect against malware. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP*, 2016.

[55] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS# 1: RSA cryptography specifications version 2.2. *Internet Engineering Task Force, Request for Comments*, 8017, 2016.

[56] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy (SP)*, 2013.

[57] National ICT Australia. UNSW Advanced Operating Systems. `https://bitbucket.org/kevinelp/unsw-advanced-operating-systems`, 2014.

[58] National ICT Australia and other contributors. seL4 Libraries. `https://github.com/seL4/seL4_libs`, 2014.

[59] National ICT Australia and other contributors. The seL4 Repository. `https://github.com/seL4/seL4`, 2014.

[60] National ICT Australia and other contributors. util_libs. `https://github.com/seL4/util_libs`, 2014.

[61] National Institute of Standards and Technology. Secure Hash Signature Standard (SHS) (FIPS PUB 180-2). 2002.

[62] National Vulnerability Database. Vulnerability summary for cve-2015-6639. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6639`, 2015.

[63] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, 2013.

[64] I. D. O. Nunes, G. Dessouky, A. Ibrahim, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Towards systematic design of collective remote attestation protocols. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[65] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security Symposium*, 2019.

[66] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems. In *International Conference On Computer Aided Design*, 2019.

[67] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. A verified architecture for proofs of execution on remote devices under full software compromise. *arXiv preprint arXiv:1908.02444*, 2019.

[68] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1991.

[69] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *European Symposium on Research in Computer Security*. Springer, 2010.

[70] D. Puri. Got milk? IoT and LoRaWAN modernize livestock monitoring. `http://www.networkworld.com/article/3118803/internet-of-things/got-milk-iot-and-lorawan-modernize-livestock-monitoring.html`.

[71] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, et al. ftpm: A software-only implementation of a TPM chip. In *25th USENIX Security Symposium*, 2016.

[72] Raspberry Pi Foundation. Raspberry Pi 2 Model B. `https://www.raspberrypi.org/products/raspberry-pi-2-model-b/`, 2015.

[73] B. Ray, S. Douglas, S. Jason, T. Stefan, W. Bryan, and W. Louis. The simon and speck families of lightweight block ciphers. Technical report, Cryptology ePrint Archive, Report./404, 2013.

[74] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 2012.

[75] M. Saarinen and J. Aumasson. The blake2 cryptographic hash and message authentication code (mac). Technical report, 2015.

[76] A.-R. Sadeghi, M. Schunter, A. Ibrahim, M. Conti, and G. Neven. SANA: Secure and scalable aggregate network attestation. In *ACM Conference on Computer and Communications Security*, 2016.

[77] E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, pages 10–20. Springer, 2004.

[78] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. SCUBA: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security (WiSe)*, 2006.

[79] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[80] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.

[81] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving*, 2011.

[82] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. In *ACM SIGPLAN Notices*, volume 48, pages 471–482. ACM, 2013.

[83] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. In *Advances in Cryptology*, 2017.

[84] Texas Instruments. MSP430-GCC-OPENSOURCE GCC - Open Source Compiler for MSP Microcontrollers. `http://www.ti.com/tool/MSP430-GCC-OPENSOURCE`, 2017.

[85] The OpenSSL Project. Openssl 1.1.0-pre7-dev. `https://github.com/openssl/openssl/`, 2016.

[86] Trusted Computing Group. Trusted platform module (tpm). `http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/`.

[87] J. Vijayan. Stuxnet renews power grid security concerns. `http://www.computerworld.com/article/2519574/security0/stuxnet-renews-power-grid-security-concerns.html`, june 2010.

[88] R. Waugh. Smart TV hackers are filming people having sex on their sofas. `http://metro.co.uk/2016/05/23/smart-tv-hackers-are-filming-people-having-sex-on-their-sofas-and-putting-it-on-porn-sites-5899248/`.

[89] Xilinx. ISE Design Suite. `https://www.xilinx.com/content/xilinx/en/downloadNav/design-tools/v2012_4---14_7.html`, 2013.

[90] W. Yan, Z. Zhang, and N. Ansari. Revealing packed malware. *ieee seCurity & PrivaCy*, 6(5):65–69, 2008.

[91] Y. Yang, X. Wang, S. Zhu, and G. Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *IEEE International Symposium on Reliable Distributed Systems*, 2007.