Title
Verifiable Integrity and Availability for Code and Execution in Simple Embedded Systems

Author
De Oliveira Nunes, Ivan

Publication Date
2021

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Verifiable Integrity and Availability for Code and Execution in Simple Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Networked Systems

by

Ivan De Oliveira Nunes

Dissertation Committee:
Dr. Gene Tsudik, Chair
Dr. Ardalan Amiri Sani
Dr. N. Asokan

2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Wow, looks like it's time to end this chapter of my life! Looking back, these past five years were unbelievable. I feel very lucky and I thank God for putting all the amazing people that I list below in my path. At this point, there's probably one thing I am sure about: I couldn't have come this far without these people. Tell me about "standing on the shoulders of giants", Google Scholar (or whoever said it originally)!

I must start by thanking the person who brought me this far, my Ph.D. advisor, Gene Tsudik. For teaching me everything I know in computing security. For offering amazing feedback and seminal ideas that were instrumental to this entire process. For genuinely caring about my success and for helping me to pursue the career that I've always dreamed of.

I would also like to thank my committee members, Ardalan Amiri Sani and N. Asokan, for their interest in my research and for taking the time to provide me with their mentorship and constructive criticism. It was a great pleasure to have such outstanding researchers on my Ph.D. committee.

In addition to my Ph.D. mentors, I could not have come this far without my early mentors and educators. I am immensely thankful to all of them for the many, many lessons. I would like to cite and especially thank my M.Sc. advisors – Antonio Loureiro and Pedro Vaz De Melo – and my Bachelor's advisors – Magnos Martinello and Roberta Lima Gomes – who were the very first to introduce me to the "research world".

I am immensely thankful to all of my co-authors. Someone once told me: "If you're not smart, you need smart co-authors." Well... guilty as charged. I do have pretty amazing co-authors! The list is too long, so I won't risk trying to name one by one. It has been really amusing and humbling to work with such a fantastic and diverse group of people. Out of these many co-authors, I especially thank Norrathep Rattanavipanon (who worked along with me in major parts of this dissertation), Sashidhar Jakkamsetti, Seoyeon Hwang, Chris Wood, and Esmerald Aliaj. These were/are also Ph.D. students at UCI who were directly involved on some of the hands-on parts of my Ph.D. research.

Out of the friends I've made during my five years at UCI, I couldn't forget to mention the Sprout Lab members that overlapped with my generation: Norrathep Rattanavipanon, Ercan Ozturk, Tyler Kaczmarek, Chris Wood, Yoshimichi Nakatsuka, Seoyeon Hwang, Sashidhar Jakkamsetti, and Gene Tsudik. Thanks for all the fun times together! I hope to see you around very soon!

Sorry if this is starting to take too long, but outside UCI, the list of personal friends and family that deserve my gratitude isn't small either. I am truly blessed to have all of you in my life.

First and foremost, I thank my wife, Renata, for her unconditional love and support. For being there for me at every step of the way. For being my best friend. I am so proud of what we've achieved together so far and I can't wait to start this new chapter of our lives.

I thank my parents – Joao and Angelica – and my brother – Igor. I've always felt all of you close to my heart, no matter how physically distant. You've been my strongest and longest-term supporters. You gave me everything. Knowing that you were there for me and seeing your genuine joy in each and every one of my accomplishments (no matter how small) is what always kept me moving forward.

Finally, I thank all my family members – especially my grandparents (Nilo, Venina, Rolando, and Marina), parents in law (Renato and Rosangela), my sister-in-law (Fernanda), and all my many uncles, aunts, and cousins. I am also thankful to all my friends – especially from Darwin, CsF-Irvine, UFES-09, and WiseMap-UFMG. Thank you for your best wishes and for staying in touch despite the distance!

CURRICULUM VITAE

Ivan De Oliveira Nunes

EDUCATION

Ph.D. in Networked Systems — **2021**
University of California, Irvine — *Irvine, California*

M.S. in Computer Science — **2016**
Universidade Federal de Minas Gerais — *Belo Horizonte, MG, Brazil*

B.S. in Computer Engineering — **2014**
Universidade Federal do Espirito Santo — *Vitoria, ES, Brazil*

PROFESSIONAL EXPERIENCE

Assistant Professor — **2021–Current**
Rochester Institute of Technology — *Rochester, New York*

Graduate Research Assistant — **2016–2021**
University of California, Irvine — *Irvine, California*

Summer Associate (Ph.D. Intern) — **Summer 2017, Spring & Summer 2018**
Visa Research — *Palo Alto, California*

Ph.D. Intern — **Summer 2019, Summer 2020**
SRI International — *Menlo Park, California*

PAPERS IN SUBMISSION OR UNDER REVIEW

Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. **TAROT : Trigger-based Active Root-Of-Trust (for Tiny Embedded Devices).** Undergoing Major Revisions to (conditionally) appear at USENIX Security Symposium. 2022.

REFEREED CONFERENCE PUBLICATIONS

Mahmoud Ammar, Bruno Crispo, Ivan De Oliveira Nunes, and Gene Tsudik. **Delegated Attestation: Scalable Remote Attestation of Commodity CPS by Blending Proofs of Execution with Software Attestation.** ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec). 2021.

Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. **On the TOCTOU Problem in Remote Attestation.** ACM Conference on Computer and Communications Security (CCS). 2021.

Ivan De Oliveira Nunes, Sashidhar Jakkamsetti and Gene Tsudik. **DIALED: Data Integrity Attestation for Low-end Embedded Devices.** Design Automation Conference (DAC). 2021.

Ivan De Oliveira Nunes, Xuhua Ding, and Gene Tsudik. **On the Root of Trust Identification Problem.** In 20th ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN). 2021.

Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. **Tiny-CFA: A Minimalistic Approach for Control Flow Attestation Using Verified Proofs of Execution.** In Design, Automation & Test in Europe Conference & Exhibition (DATE). 2021.

Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. **APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise.** In 29th USENIX Security Symposium (USENIX Security 20). 2020.

Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. **VRASED: A verified hardware/software co-design for remote attestation.** In 28th USENIX Security Symposium (USENIX Security 19), pp. 1429-1446. 2019.

Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. **Towards systematic design of collective remote attestation protocols.** In 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1188-1198. IEEE, 2019.

Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. **PURE: Using Verified Remote Attestation to Obtain Proofs of Update, Reset and Erasure in low-End Embedded Systems.** In IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1-8. 2019.

Ivan O. Nunes and Gene Tsudik. **KRB-CCN: Lightweight Authentication and Access Control for Private Content-Centric Networks.** In International Conference on Applied Cryptography and Network Security (ACNS), pp. 598-615. Springer, Cham, 2018.

Ivan De Oliveira Nunes, Karim Eldefrawy, and Tancrede Lepoint. **Secure Non-interactive User Re-enrollment in Biometrics-Based Identification and Authentication Systems.** In International Symposium on Cyber Security Cryptography and Machine Learning (CSCML), pp. 162-180. Springer, Cham, 2018.

Ivan O. Nunes, Gene Tsudik, and Christopher A. Wood. **Namespace tunnels in content-centric networks.** In 2017 IEEE 42nd Conference on Local Computer Networks (LCN), pp. 35-42. IEEE, 2017.

Ivan O. Nunes, Clayson Celes, Michael D. Silva, Pedro OS Vaz de Melo, and Antonio AF

Loureiro. **GRM: Group Regularity Mobility Model.** In Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM), pp. 85-89. 2017.

Michael D. Silva, Ivan O. Nunes, Raquel AF Mini, and Antonio AF Loureiro. **ST-Drop: A novel buffer management strategy for D2D opportunistic networks.** In 2017 IEEE Symposium on Computers and Communications (ISCC), pp. 1300-1305. IEEE, 2017.

Antonio L. Maia Neto, Artur LF Souza, Italo Cunha, Michele Nogueira, Ivan Oliveira Nunes, Leonardo Cotta, Nicolas Gentille et al. **AoT: Authentication and access control for the entire iot device life-cycle.** In Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys), pp. 1-15. 2016.

Ivan Oliveira Nunes, Pedro OS Vaz de Melo, and Antonio AF Loureiro. **Group mobility: Detection, tracking and characterization.** In 2016 IEEE International Conference on Communications (ICC), pp. 1-6. IEEE, 2016.

REFEREED JOURNAL PUBLICATIONS

Ivan De Oliveira Nunes, Karim Eldefrawy, and Tancrede Lepoint. **SNUSE: A secure computation approach for large-scale user re-enrollment in biometric authentication systems.** Future Generation Computer Systems 98 (2019): 259-273.

Ivan O. Nunes, Clayson Celes, Igor Nunes, Pedro OS Vaz de Melo, and Antonio AF Loureiro. **Combining spatial and social awareness in D2D opportunistic routing.** IEEE Communications Magazine 56, no. 1 (2018): 128-135.

Ivan O. Nunes, Clayson Celes, Pedro OS Vaz de Melo, and Antonio AF Loureiro. **GROUPS-NET: Group meetings aware routing in multi-hop D2D networks.** Computer Networks 127 (2017): 94-108.

Ivan O. Nunes, Pedro OS Vaz de Melo, and Antonio AF Loureiro. **Leveraging D2D multihop communication through social group meeting awareness.** IEEE Wireless Communications 23, no. 4 (2016): 12-19.

ABSTRACT OF THE DISSERTATION

Verifiable Integrity and Availability for Code and Execution in Simple Embedded Systems

By

Ivan De Oliveira Nunes

Doctor of Philosophy in Networked Systems

University of California, Irvine, 2021

Dr. Gene Tsudik, Chair

Modern society is increasingly surrounded by, and is growing accustomed to, a wide range of Cyber-Physical Systems (CPS), Internet-of-Things (IoT), and smart devices. They often perform safety-critical functions, e.g., personal medical devices, automotive CPS as well as industrial and residential automation (such as sensor-alarm combinations). On the lower end of the scale, these devices are small, cheap, and specialized sensors and/or actuators. They tend to host small CPUs, have small amounts of memory, and run simple software. If such devices are left unprotected, consequences of forged sensor readings or ignored actuation commands can be catastrophic, particularly, in safety-critical settings. This prompts the following questions: (1) How to trust data produced, or guarantee that actions will be performed, by a simple remote embedded device?, (2) How to bind actions and results to the execution of expected software? and, (3) Can (1) and (2) be attained even if all software on a device can be modified and/or compromised (e.g., by malware) at any given time?

This dissertation presents a set of hardware/software co-designs for obtaining several security services – namely remote attestation, TOCTOU-avoidance, proofs of execution, and root of trust availability – which can be used to assure the integrity and availability of software and its execution, even on some of the most resource-constrained micro-controllers. We realize these services with four formally verified and publicly available architectures (VRASED,

RATA, APEX, and TAROT) and show how they have been securely implemented atop the TI MSP430 micro-controller at a relatively low-cost.

Chapter 1

Introduction

The number and diversity of special-purpose computing devices has been increasing dramatically. This includes all kinds of embedded devices, cyber-physical systems (CPS) and Internet-of-Things (IoT) gadgets, utilized in various "smart" or instrumented settings, such as homes, offices, factories, automotive systems, and public venues. Tasks performed by these devices are often safety-critical. For example, a typical industrial control system depends on physical measurements (e.g., temperature, pressure, humidity, speed) reported by sensors, and on actions taken by actuators, such as: turning on the A/C, sounding an alarm, or reducing speed.

A cyber-physical control system is usually composed of multiple sensors and actuators, at the core of each is a micro-controller unit (MCU), typically running simple software, often on "bare metal", i.e., with no microkernel or hypervisor. They tend to be operated by a remote central control unit and despite their potential importance to overall system functionality, low-end devices are typically designed to minimize cost, physical size and energy consumption.

A compromised MCU can spoof sensed quantities or ignore actuation commands, leading to

potentially catastrophic results. For example, in a *smart* city, large-scale erroneous reports of electricity consumption by smart meters might lead to power outages. A medical device that returns incorrect values when queried by a remote physician might result in a wrong drug being prescribed to a patient. A compromised car engine temperature sensor that reports incorrect (low) readings can lead to undetected overheating and major damage. Unfortunately, these examples are not theoretical or hypothetical. Actuation devices have been abused by malware to impact both security and safety in the Stuxnet [111] case. Whereas malware on sensors can undermine privacy by obtaining ambient information [51]. Furthermore, clever malware can turn vulnerable IoT devices into zombies that can become sources for DDoS attacks. For example, in 2016, a multitude of compromised *smart* cameras and DVRs formed the Mirai Botnet [12] which was used to mount a massive-scale DDoS attack (the largest in history). However, despite very real risks of remote software compromise, most users believe that these devices execute expected software and thus perform their expected function.

At the lower-end of the spectrum, MCUs are designed with extremely strict constraints on monetary cost, physical size, and energy consumption (e.g, TI MSP430[1] and Atmel ATMega AVR[2]). Therefore, it is unrealistic to expect such devices, by themselves, to prevent malware infection via sophisticated security mechanisms (e.g., similar to those available on laptops, smartphones, and other types of higher-end embedded devices).

To address this problem, several tiny Roots-of-Trust (RoTs) [49, 87, 10, 22, 68] were proposed to enable Remote Attestation (RA), i.e., remote verification of an embedded device's software state. These tiny RoTs are typically designed as hardware/software (hybrid) co-designs, aiming to achieve the same level of security of more expensive hardware-based architectures (see Chapter 2 for details) at much lower hardware cost. Despite substantial progress over the past decade, to the best of our knowledge, prior architectures have the following limitations:

[1] `http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/applications.html`
[2] `https://www.microchip.com/design-centers/8-bit/avr-mcus`

1. Little or no attention has been devoted to formal verification. In particular, no hybrid RA designs and implementations have been formally verified with respect to claimed security properties. We argue that the high-assurance and rigor derivable from utilizing computer-aided formal verification to guarantee security of the design and implementation of RA techniques can increase RA robustness and thus its potential for practical adoption.

2. Prior RA techniques verify the remote device at the time when RA functionality is executed, thus providing no information about the state of the device before the current RA execution or between consecutive RA executions. Therefore, presence of transient malware that infects the device and leaves prior to the next RA instance can not be detected. This important problem, called Time-Of-Check-Time-Of-Use (TOCTOU), is well-known in the research literature and remained unaddressed in the context of hybrid RA.

3. Prior techniques can be used to prove that a given binary is present on a remote device, but cannot prove successful execution of that binary. They are also cannot bind claimed execution outputs/results (e.g., a sensed value) to the correct execution of the attested binary (e.g., a sensing task).

4. All prior techniques *operate reactively*. Therefore, they can not *guarantee* that a desired action *will be performed*, since malware controlling the device can trivially block access to the RoT by ignoring/discarding received commands. This is a major and important problem because it allows malware to effectively "brick" or incapacitate a potentially huge number of (possibly mission-critical) devices.

In this dissertation we address these limitations by presenting four core architectures. Specifically, we describe the design, implementation, and formal verification of:

1. *VRASED*: the first formally verified hybrid RA architecture.

2. *RATA*: the first hybrid RA architecture secure against TOCTOU attacks.

3. *APEX*: an architecture for generating unforgeable proofs of software execution in low-end embedded systems.

4. *TAROT*: an active RoT architecture, which guarantees that safety-critical tasks are always performed upon specific trigger-s, despite compromise of the MCU application software.

1.1 Dissertation Structure

Chapter 2 overviews concepts that are relevant to the dissertation as a whole, including its scope, the foundational security service – Remote Attestation (RA) – and the proof strategy and verification methodology used to argue security of architectures proposed in the following chapters. Chapter 3 presents *VRASED*: a verified hardware/software co-design for remote attestation targeting low-end embedded devices. Chapter 4 provides a systematic treatment and a formal definition for the Time-Of-Check Time-Of-Use (TOCTOU) problem in RA, as well as *RATA*: a hardware component that makes hybrid RA techniques TOCTOU-Secure (e.g., it detects presence of transient malware). Chapter 5 focuses on the problem of proving correct execution of attested software in a low-end embedded device, a functionality referred to as Proof of Execution (PoX). This chapter presents *APEX*: a formally verified PoX architecture built atop (and securely composed with) *VRASED*. Next, Chapter 6 explores guaranteed execution of expected safety-critical tasks/actions on low-end MCUs. It introduces TAROT: a trigger-based active root of trust that is also formally specified and verified.

We note that Chapter 2 only includes background on topics relevant to the dissertation as a whole. Background specific to a particular chapter is provided within that chapter.

Similarly, system and adversary models are defined on a per-chapter basis. For the most part, notation is consistent across chapters.

Chapter 2

Background

Abstract

This chapter overviews background concepts for dissertation. We start, in Section 2.1, by defining the scope of targeted devices. We motivate this choice and discuss some general intended contributions (specific contributions are outlined in subsequent chapters). Next, in Section 2.2, we overview Remote Attestation (RA): a security service that enables verification of the software state of a potentially compromised remote device – a prover ($\mathcal{P}$rv) – by a trusted verifier ($\mathcal{V}$rf). RA also serves as a foundation to many other security services. We review RA in detail since one of the contributions presented in this dissertation is VRASED (Chapter 3): a formally verified RA architecture. Additionally, Chapter 4 discusses an important RA security feature (RA TOCTOU-Security) and Chapter 5 develops a verified architecture for proofs of execution that is built upon RA. Lastly, in Section 2.3, we present the formal verification methodology and proof strategy used across the remaining chapters to prove security of the proposed security architectures at both protocol and implementation levels. We note that this chapter only overviews concepts that are relevant to the dissertation as a whole. Concepts and definitions that are only relevant or applicable to a particular chapter are discussed within that chapter.

2.1 Scope: Low-end Embedded Devices

This work focuses on CPS/IoT sensors and actuators (or hybrids thereof) with low computing power. These are some of the smallest and weakest devices based on low-power single-core Micro-Controller Units (MCUs) with only a few KBytes of program and data memory. Two prominent examples are: Atmel AVR ATmega and TI MSP430: 8- and 16-bit CPUs, typically running at 1-16MHz clock frequencies, with $\approx$ 64 KBytes of addressable memory. SRAM is used as data memory with the size normally ranging between 4 and 16KBytes, while the rest of address space is available for program memory. They have neither Memory Management Units (MMUs) to support virtual memory nor Memory Protection Units (MPUs) to control access to any parts of memory. They also lack architectural support for privileged/exception layers. Therefore, such devices usually run software atop "bare metal" and execute instructions in place (physically from program memory).

In terms of practicality and applicability, we believe that a security architecture suitable for these lowest-end MCU-s could be adapted (and potentially enriched) for higher-end devices with larger hardware budgets, while the other direction is more challenging. In addition, simpler devices are easier to model and reason about formally. Hence, they represent a good starting point for the design and verification of provably secure architectures such as the ones proposed in this work.

Our implementations are based on MSP430. This choice is due to public availability of a well-maintained open-source MSP430 hardware design from Open Cores [56]. Nevertheless, we believe that our machine models and methodology are applicable to other low-end MCUs in the same class, such as Atmel AVR ATmega. We also hope that some lessons learned in this work can be useful to design and prove security of similar services targeting higher-end devices in the future.

2.2 Attestation in Low-end Devices

Attestation facilitates detection of malware presence on a remote device. Specifically, it allows a trusted verifier ($\mathcal{V}\mathsf{rf}$) to remotely measure the software state of an untrusted remote device ($\mathcal{P}\mathsf{rv}$). As shown in Figure 2.1, attestation is typically obtained via a simple challenge-response protocol:

1. $\mathcal{V}\mathsf{rf}$ sends an attestation request containing a challenge ($\mathcal{C}\mathsf{hal}$) 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 the attestation request and computes an *authenticated integrity check* over its memory and $\mathcal{C}\mathsf{hal}$. The memory region might be either pre-defined, or explicitly specified in the request.
3. $\mathcal{P}\mathsf{rv}$ reports the result to $\mathcal{V}\mathsf{rf}$.
4. $\mathcal{V}\mathsf{rf}$ receives the result from $\mathcal{P}\mathsf{rv}$, and checks whether it corresponds to a valid memory state.

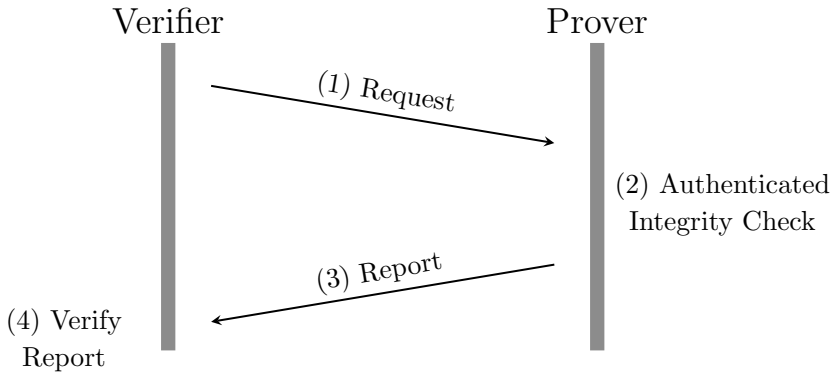


Figure 2.1: Attestation interaction

The *authenticated integrity check* can be realized as a Message Authentication Code (MAC) over $\mathcal{P}\mathsf{rv}$'s memory. Computing a MAC requires $\mathcal{P}\mathsf{rv}$ to have a unique secret key, denoted by $\mathcal{K}$ – either a symmetric key shared with $\mathcal{V}\mathsf{rf}$, or a private key for which the corresponding public key is known to $\mathcal{V}\mathsf{rf}$. In this dissertation we assume the former, noting that security differences between the two cases are small. This $\mathcal{K}$ must reside in secure storage, where it is

not accessible to any software running on $\mathcal{P}$rv, except for trusted and typically immutable attestation code (or attestation hardware engine, when present). Since most RA threat models assume a fully compromised software state on $\mathcal{P}$rv, secure storage implies some level of hardware support. Most attestation techniques fall into three groups: software-based, hardware-based, and hybrid.

Software-based (or timing-based) attestation is the only viable approach for legacy devices with no hardware security features. Without hardware support, it is (currently) impossible to guarantee that $\mathcal{K}$ is not accessible by malware that fully compromises $\mathcal{P}$rv software state. Therefore, security of software-based approaches [98, 76] is attained by setting strict thresholds for communication delays between $\mathcal{V}$rf and $\mathcal{P}$rv (instead of relying on cryptographic secrets). Thus, software-based attestation is unsuitable for multi-hop and jitter-prone communication (not remote), or settings where a compromised $\mathcal{P}$rv is aided (during attestation) by a more powerful accomplice device. It also requires strong constraints and assumptions on the hardware platform and attestation usage [71, 75].

Hardware-based approaches, on the other extreme, enable attestation to happen remotely but require either i) $\mathcal{P}$rv's attestation functionality to be housed entirely within dedicated hardware, e.g., Trusted Platform Modules (TPMs) [109]; or ii) modifications to the CPU semantics or instruction sets to support the execution of trusted software, e.g., SGX [64] or TrustZone [14]. Such hardware features are too expensive (in terms of physical area, energy consumption, and actual cost) for low-end devices. SANCUS [87] developed a hardware-based RA architecture specifically targeting low-end devices (MSP430, in particular). However, its footprint is heavy, increasing the total hardware cost by more than 100% that of the unmodified MSP430 CPU core.

While neither hardware- nor software-based approaches are well-suited for settings where low-end devices are not directly connected (e.g, remote IoT settings), **hybrid** RA (based on HW/SW co-design) is a more promising approach. **Hybrid** RA aims at providing the

same security guarantees as hardware-based techniques with minimal hardware support. SMART [49] is the first hybrid RA architecture targeting low-end MCUs. In SMART, the authenticated integrity check is implemented in software. SMART's small hardware footprint guarantees that the attestation code runs safely and that the attestation key is not leaked. HYDRA [48] is a hybrid RA scheme that relies on a secure boot hardware feature and on a formally verified secure micro-kernel. Trustlite [68] modifies MPU and CPU exception engine hardware to implement RA on the Intel Siskiyou Peak research platform [108]. Tytan [22] is built on top of Trustlite, extending its capabilities for applications with real-time requirements.

Most RA architectures (including the ones mentioned thus far) focus on measuring a snapshot of $\mathcal{P}$rv's memory at the time of the attestation computation[1]. As such, they are useful to measure executables, i.e., to prove to $\mathcal{V}$rf that a particular binary is currently installed on $\mathcal{P}$rv. however, they are usually insufficient to prove execution or execution properties. For example, they can not prove that the attested binary is ever executed. Additionally, consider runtime/data-oriented attacks [106] that tamper with execution state on the program's stack or heap to arbitrarily divert the program's execution flow and/or corrupt computation results. Such attacks need not modify the executable itself. Thus, they are not detectable by RA alone. Runtime attacks can be launched by a variety of means. For instance, in languages such as C, C++, and Assembly (which are widely used to program MCU-s), buffer overflows [39] can overwrite functions' return addresses, hijacking the program's control-flow and launching well-known Return-Oriented Programming (ROP) attacks [102]. These attacks are especially dangerous for low-end MCU-s that can not avail themselves of more sophisticated OS-based mitigations, e.g., canaries, Address Space Layout Randomization (ASLR), and Control-Flow Integrity (CFI) techniques, available in high-end platforms.

[1]We note that the term RA may be used more broadly to refer to other forms of attestation (e.g., content attestation). In the context of this dissertation, RA reffers specifically to a service used to measure memory in remote devices.

Runtime attestation [6, 45, 44, 116, 43, 105, 91], including both control-flow attestation (CFA) and data-flow attestation (DFA), augments RA capability to enable detection of control-flow and data-only attacks. In a nutshell, CFA techniques provide $\mathcal{V}$rf with a report that allows it to not only learn if the expected code is loaded on $\mathcal{P}$rv, but also which particular instruction path was taken during each execution of this program. In other words, CFA provides $\mathcal{V}$rf with an authentic and unforgeable report that allows $\mathcal{V}$rf to learn if instructions of a given program were executed in a particular expected/legal order, or a set thereof. This is typically achieved by securely logging information associated with the destination of each control-flow altering instruction, e.g., `jumps`, `branches`, `returns`, during program execution. Similarly, DFA allows $\mathcal{V}$rf to detect whether intermediate computation values (e.g., stored in local variables) were corrupted throughout the attested program's execution by securely logging modifications to these variables into an authenticated report log.

Prior CFA and DFA techniques have been implemented on medium- to high-end embedded devices (e.g., Raspberry Pi, and RISC-V based processors), by leveraging trusted hardware support, such as ARM TrustZone (in [6, 105]) or hardware branch monitors/hardware hash engines (in [45, 44, 116]). However, for the lowest-end MCU-s, these requirements are too costly, as their hardware overhead is often higher than the total cost of the MCU's core itself, in terms of size, energy and monetary cost.

Despite much progress, a major missing aspect in RA research is high-assurance and rigor obtained by using formal methods to guarantee security of a concrete RA design and its implementation (hardware and software). We believe that verifiability and formal security guarantees are particularly important for hybrid RA designs aimed at low-end embedded and IoT devices, as hybrid designs are hard to implement correctly and their proliferation keeps growing. This serves as the main motivation for our efforts to develop a formally verified RA architecture (Chapter 3) and subsequent verified TOCTOU-Secure RA architecture (Chapter 4) which enhances hybrid RA against migratory and self-relocating malware, as well as

makes RA substantially more efficient.

Given the lack of suitable approaches to prove execution properties in low-end devices, our work also proposes the development and formal verification of an architecture for proofs of execution (Chapter 5) in low-end devices. In subsequent work [43, 91], this architecture was also shown to be sufficient, as the only hardware feature, to obtain CFA and DFA.

Finally, we observe that the aforementioned architectures focus on the problem of "proving integrity", i.e., they offer proofs for $\mathcal{P}$rv's state or that actions (e.g., execution, updates, erasure & reset/reboot [41, 9, 15]) have happened. However, they can not guarantee that actions/commanded tasks will be performed by a low-end (and potentially compromised) $\mathcal{P}$rv. This problem – related to **availability** – is the focus of Chapter 6.

2.3 Linear Temporal Logic, Model Checking, and Formal Verification

Computer-aided formal verification typically involves three basic steps. First, the system of interest (e.g., hardware, software, communication protocol) is described using a formal model, e.g., a Finite State Machine (FSM). Second, properties that the model should satisfy are formally specified. Third, the system model is checked against formally specified properties to guarantee that the system retains them. This can be achieved by either Theorem Proving [79] or Model Checking [36]. In this work, we use the latter to verify the implementation of system sub-modules, and the former to prove new properties derived from the combination (conjunction) of machine model axioms and sub-properties that were proved for the implementation of individual sub-modules.

In one instantiation of model checking, properties are specified as *formulae* using Linear

Temporal Logic (LTL) and system models are represented as FSMs. Hence, a system is represented by a triple (S, S_0, T), where S is a finite set of states, $S_0 \subseteq S$ is the set of possible initial states, and $T \subseteq S \times S$ is the transition relation set – it describes the set of states that can be reached in a single step from each state. The use of LTL to specify properties allows representation of expected FSM behavior over time.

Our verification strategy applies the widely used model checker NuSMV [34], geared for verifying generic HW or SW models. For digital hardware described at Register Transfer Level (RTL) – which is the case in this work – conversion from Hardware Description Language (HDL) to NuSMV models is simple. Furthermore, it can be automated [65], because the standard RTL design already relies on describing hardware as an FSM. We rely on this automated verification pipeline to verify several sub-modules of all architectures discussed in this dissertation. Details of this pipeline are presented in Chapter 3.

LTL specifications are is particularly useful for verifying sequential systems. In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL extends propositional logic with **temporal quantifiers**, thus enabling sequential reasoning. In this work, we are interested in the following LTL quantifiers:

- $\mathbf{X}\phi$ – ne<u>X</u>t ϕ: holds if ϕ is true at the next system state.
- $\mathbf{F}\phi$ – <u>F</u>uture ϕ: holds if there exists a future state where ϕ is true.
- $\mathbf{G}\phi$ – <u>G</u>lobally ϕ: holds if for all future states ϕ is true.
- ϕ $\mathbf{U}$ ψ – ϕ <u>U</u>ntil ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.
- ϕ $\mathbf{W}$ ψ – ϕ <u>W</u>eak until ψ: holds if, assuming a future state where ψ holds, ϕ holds for all states prior to that. If ψ never becomes true, ϕ must hold forever. Or, more formally: $\phi \mathbf{W} \psi \equiv (\phi \mathbf{U} \psi) \vee \mathbf{G}(\phi)$
- ϕ $\mathbf{B}$ ψ – ϕ <u>B</u>efore ψ: holds if the existence of state where ψ holds implies the existence of an earlier state where ϕ holds. This temporal quantifier can be expressed using $\mathbf{U}$

through the equivalence: $\phi \ \mathbf{B} \ \psi \equiv \neg(\neg\phi \ \mathbf{U} \ \psi)$.

This set of temporal connectives combined with propositional connectives (with their usual meanings) allows us to specify powerful rules. NuSMV works by checking LTL specifications against the system FSM for all reachable states in such FSM.

A model checker (NuSMV in our case) performs proofs through automated and exhaustive enumeration of all possible system states. If the desired specification is found not to hold for specific states (or transitions between states), a trace of the model that leads to the erroneous state is provided, and the implementation can then be fixed accordingly. As a consequence of exhaustive enumeration, proofs for complex systems that involve complex properties often do not scale well due to so-called "state explosion" problem.

To cope with this problem, our verification approach is to specify smaller LTL sub-properties separately and verify each respective sub-module for compliance. In this process, our verification pipeline automatically converts digital hardware, described at RTL using Verilog, to Symbolic Model Verifier (SMV) [85] FSMs using Verilog2SMV [65] (see Chapter 3 for details). The SMV representation is then fed to NuSMV [35] for verification. Finally, the composition of the LTL sub-properties (verified in the model checking phase) is proven to achieve the end-to-end implementation goals (which are more complex LTL statements) of the particular system using an LTL theorem prover [47]. The verification strategy is depicted in Figure 2.2.

Depending on the security service, end-to-end goals for the implementation may themselves capture the overall security definition (e.g., Chapter 6) for the system. In other cases, especially when security depends on cryptographic assumptions, end-to-end implementation goals are required so that a cryptographic security proof (e.g., cryptographic reduction) holds. The latter is the case in Chapters 3, 4, and 5.

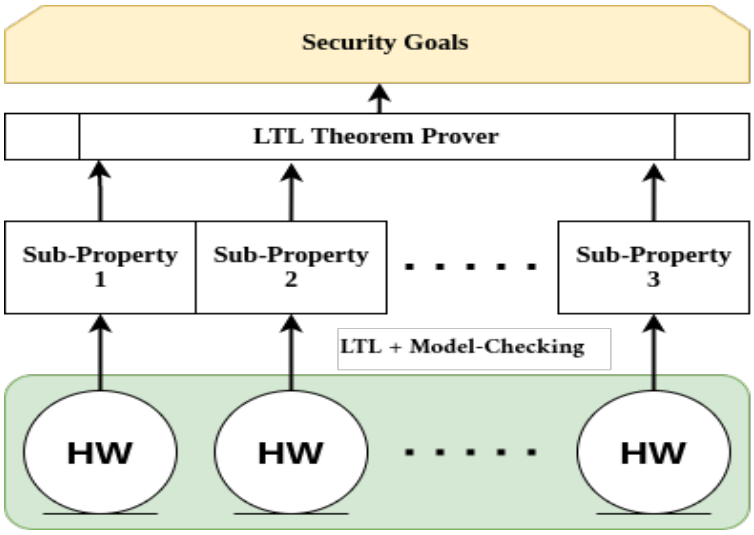


Figure 2.2: Overall Verification strategy

Chapter 3

VRASED: Verifiable Remote Attestation for Simple Embedded Systems

Abstract

This chapter presents the first step towards formal verification of RA. We design and verify an architecture called *VRASED*: **V**erifiable **R**emote **A**ttestation for **S**imple **E**mbedded **D**evices. *VRASED* instantiates a hybrid (HW/SW) RA co-design aimed at low-end embedded systems, e.g., simple IoT devices. *VRASED* provides a level of security comparable to HW-based approaches, while relying on SW to minimize additional HW costs. Since security properties must be jointly guaranteed by HW and SW, verification is a challenging task. To demonstrate *VRASED*'s practicality and low overhead, we instantiate and evaluate it on a commodity platform (TI MSP430). *VRASED* was deployed using the Basys3 Artix-7 FPGA and its implementation is publicly available. The contributions described in this chapter appeared in the Proceedings of the USENIX Security Symposium – 2019 (see [40]).

3.1 Introduction

Even though numerous RA techniques with different assumptions, security guarantees, and designs, have been proposed [97, 92, 76, 49, 68, 22, 48, 50, 87, 89, 31, 32, 61, 87, 29], a major missing aspect of RA is the high-assurance and rigor derivable from utilizing computer-aided formal verification to guarantee security of the design and implementation of RA techniques. Because all aforementioned architectures and their implementations are not systematically designed from abstract models, their soundness and security can not be formally argued. In fact, our RA verification efforts revealed that a previous hybrid RA design – SMART [49] – assumed that disabling interrupts is an atomic operation and hence opened the door to compromise of $\mathcal{P}$rv's secret key in the window between the time of the invocation of disable interrupts functionality and the time when interrupts are actually disabled. Another low/medium-end architecture – Trustlite [68] – does not achieve our formal definition of RA soundness. As a consequence, this architecture is vulnerable to self-relocating malware (See [27] for details). Formal specification of RA properties and their verification significantly increases our confidence that such subtle issues are not overlooked.

In this work we take a "verifiable-by-design" approach and develop, from scratch, an architecture for **V**erifiable **R**emote **A**ttestation for **S**imple **E**mbedded **D**evices (*VRASED*). *VRASED* is the first formally specified and verified RA architecture accompanied by a formally verified implementation. Verification is carried out for all trusted components, including hardware, software, and the composition of both, all the way up to end-to-end notions for RA soundness and security. The resulting verified implementation – along with its computer proofs – is publicly available [1]. Formally reasoning about, and verifying, *VRASED* involves overcoming major challenges not previously encountered in the context of RA:

1 – Formal definitions of: `(i)` end-to-end notions for RA soundness and security; `(ii)` a realistic machine model for low-end embedded systems; and `(iii)` *VRASED*'s guarantees.

These definitions must be made within a single formal system that is powerful enough to provide a common ground for reasoning about their interplay. In particular, our end goal is to prove that the definitions of RA soundness and security are implied by *VRASED*'s guarantees when applied to our machine model. As discussed in Chapter 2, our formal system of choice is Linear Temporal Logic (LTL).

2 – Automatic end-to-end verification of complex systems such as *VRASED* is challenging from the computability perspective, as the space of possible states is extremely large. To cope with this challenge, we take a "divide-to-conquer" approach. We start by dividing the end-to-end goal of RA soundness and security into smaller sub-properties that are also defined in LTL. Each HW sub-module, responsible for enforcing a given sub-property, is specified as a Finite State Machine (FSM), and verified using a Model Checker. *VRASED*'s SW relies on an F* verified implementation (see Section 3.3.3) which is also specified in LTL. This modular approach allows us to efficiently prove sub-properties enforced by individual building blocks in *VRASED*.

3 – All proven sub-properties must be composed in order to reason about RA security and soundness of *VRASED*. To this end, we use a theorem prover to show (by using LTL equivalences) that the sub-properties that were proved for each of *VRASED*'s sub-modules, when composed, imply the end-to-end definitions of RA soundness and security. This modular approach enables efficient system-wide formal verification.

3.2 Overview of VRASED

VRASED is composed of a HW module (`HW-Mod`) and a SW implementation (`SW-Att`) of $\mathcal{P}$rv's behavior according to the RA protocol. `HW-Mod` enforces access control to $\mathcal{K}$ in addition to secure and atomic execution of `SW-Att` (these properties are discussed in detail below).

HW-Mod is designed with minimality in mind. The verified FSMs contain a minimal state space, which keeps hardware cost low. SW-Att is responsible for computing an attestation report. As *VRASED*'s security properties are jointly enforced by HW-Mod and SW-Att, both must be verified to ensure that the overall design conforms to the system specification.

3.2.1 Adversary Capabilities & Verification Axioms

We consider an adversary, $\mathcal{A}$dv, that can control the entire software state, code, and data of $\mathcal{P}$rv. $\mathcal{A}$dv can modify any writable memory and read any memory that is not explicitly protected by access control rules, i.e., it can read anything (including secrets) that is not explicitly protected by HW-Mod. It can also re-locate malware from one memory segment to another, in order to avoid being detected. $\mathcal{A}$dv may also have full control over all Direct Memory Access (DMA) controllers on $\mathcal{P}$rv. DMA allows a hardware controller to directly access main memory (e.g., RAM, flash or ROM) without going through the CPU.

We focus on attestation functionality of $\mathcal{P}$rv; verification of the entire MCU architecture is beyond the scope of this work. Therefore, we assume the MCU architecture strictly adheres to, and correctly implements, its specifications. In particular, our verification approach relies on the following simple axioms:

- **A1 - Program Counter:** The program counter (PC) always contains the address of the instruction being executed in a given cycle.
- **A2 - Memory Address:** Whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, and for a write access, a data write-enable bit (W_{en}) must be set.
- **A3 - DMA:** Whenever a DMA controller attempts to access main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being

accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (logical zero).

- **A4 - MCU reset:** At the end of a successful *reset* routine, all registers (including PC) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware; thus, reset handling routine can not be modified.
- **A5 - Interrupts:** When interrupts happen, the corresponding *irq* signal is set.

`SW-Att` uses the HACL* [117] HMAC-SHA256 function which is implemented and verified in F*[1]. F* can be automatically translated to C and the proof of correctness for the translation is provided in [94]. However, even though efforts have been made to build formally verified C compilers (CompCert [74] is the most prominent example), there are currently no verified compilers targeting lower-end MCUs, such as MSP430. Hence, we assume that the standard compiler can be trusted to semantically preserve its expected behavior, especially with respect to the following:

- **A6 - Callee-Saves-Register:** Any register touched in a function is cleaned by default when the function returns.
- **A7 - Semantic Preservation:** Functional correctness of the verified HMAC implementation in C, when converted to assembly, is semantically preserved.

Remark: *Axioms **A6** and **A7** reflect the corresponding compiler specification (e.g., msp430-gcc).*

Invasive physical hardware attacks are out of scope in this work. Specifically, $\mathcal{A}\mathsf{dv}$ can not modify code stored in `ROM`, induce hardware faults, or retrieve $\mathcal{P}\mathsf{rv}$ secrets via physical presence side-channels. Protection against physical attacks is considered orthogonal and could be supported via standard tamper-resistance techniques [95]. Non-invasive attacks that require physical presence, such as reprogramming the MCU software using appropriate

[1] `https://www.fstar-lang.org/`

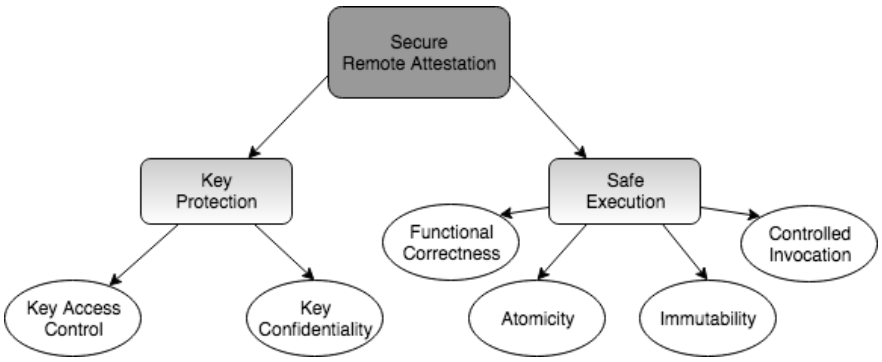


Figure 3.1: Properties of secure RA.

interfaces (without tampering with hardware), are still considered within our scope.

3.2.2 Secure RA Properties at a High-Level (Informally)

We now describe, in high level, the sub-properties required for RA. In section 3.3, we formalize these sub-properties in LTL and provide single end-to-end definitions for RA soundness and security. Then we prove that *VRASED*'s design satisfies the aforementioned sub-properties and that the end-to-end definitions for soundness and security are implied by them. The properties, shown in Figure 3.1, fall into two groups: *key protection* and *safe execution*.

Key Protection:

As mentioned earlier, $\mathcal{K}$ must not be accessible by regular software running on $\mathcal{P}$rv. To guarantee this, the following features must be correctly implemented:

- **P1- Access Control:** $\mathcal{K}$ can only be accessed by `SW-Att`.
- **P2- No Leakage:** Neither $\mathcal{K}$ (nor any function of $\mathcal{K}$ other than the correctly computed HMAC) can remain in unprotected memory or registers after execution of `SW-Att`.
- **P3- Secure Reset:** Any memory holding any function of $\mathcal{K}$ and all registers (includ-

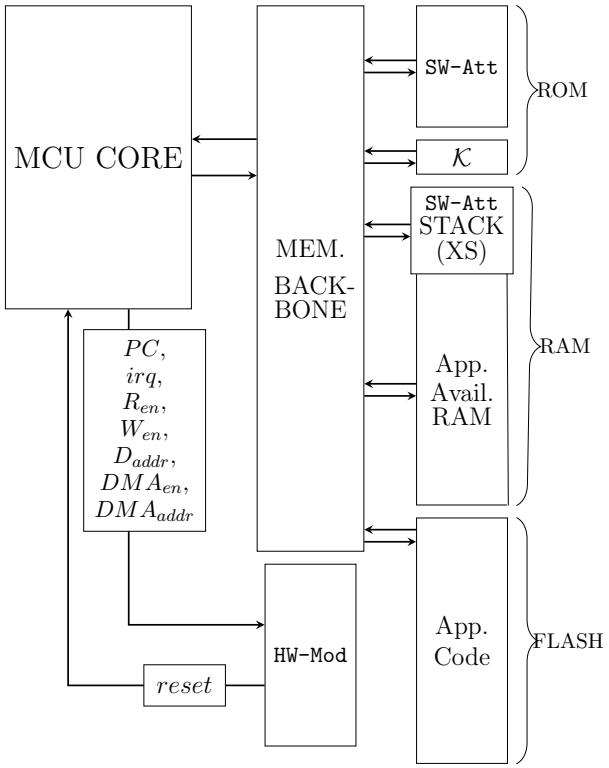


Figure 3.2: VRASED system architecture

ing PC) must be erased (or be inaccessible to regular software) after MCU reset. Since a reset might be triggered during `SW-Att` execution, lack of this property could result in leakage of privileged information about the system state or $\mathcal{K}$. Erasure of registers as part of the reset ensures that no state from a previous execution persists. Therefore, the system must return to the default initialization state.

Safe Execution:

Safe execution ensures that $\mathcal{K}$ is properly and securely used by `SW-Att` for its intended purpose. Safe execution is divided into four sub-properties:

- **P4- Functional Correctness:** `SW-Att` must implement expected behavior of $\mathcal{P}$rv's role in the RA protocol. For instance, if $\mathcal{V}$rf expects a response containing an HMAC of memory in address range $[A, B]$, `SW-Att` implementation should always reply accord-

ingly. Moreover, SW-Att must always finish in finite time, regardless of input size and other parameters.

- **P5- Immutability:** SW-Att executable must be immutable. Otherwise, malware residing in $\mathcal{P}$rv could modify SW-Att, e.g., to always generate valid RA measurements or to leak $\mathcal{K}$.
- **P6- Atomicity:** SW-Att execution can not be interrupted. The first reason for atomicity is to prevent leakage of intermediate values in registers and SW-Att's data memory (including locations that could leak functions of $\mathcal{K}$) during SW-Att execution. This relates to **P2** above. The second reason is to prevent roving malware from relocating itself to escape being measured by SW-Att.
- **P7- Controlled Invocation:** SW-Att must always start from the first instruction and execute until the last instruction. Even though correct implementation of SW-Att is guaranteed by **P4**, isolated execution of chunks of a correctly implemented code could lead to catastrophic results. Potential ROP attacks could be constructed using gadgets of SW-Att (which, based on **P1**, have access to $\mathcal{K}$) to forge valid attestation results or leak $\mathcal{K}$.

Beyond aforementioned core security properties, in some settings, $\mathcal{P}$rv might need to authenticate $\mathcal{V}$rf's attestation requests in order to mitigate potential DoS attacks on $\mathcal{P}$rv [50]. This functionality is also provided (and verified) as an optional feature in the design of *VRASED*. The differences between the standard design and the one with support for $\mathcal{V}$rf authentication are discussed in Appendix 3.9.

3.2.3 System Architecture

VRASED architecture is depicted in Figure 3.2. *VRASED* adds HW-Mod to the MCU architecture, e.g., MSP430. MCU memory layout is extended to include Read-Only Memory (ROM)

that houses SW-Att code and $\mathcal{K}$ used in the HMAC computation. Because $\mathcal{K}$ and SW-Att code are stored in ROM, we have guaranteed immutability, i.e., **P5**. *VRASED* also reserves a fixed part of the memory address space for SW-Att stack. This amounts to $\approx 3\%$ of the address space, as discussed in Section 3.5 [2]. Access control to dedicated memory regions, as well as SW-Att atomic execution are enforced by HW-Mod. The memory backbone is extended to support multiplexing of the new memory regions. HW-Mod takes 7 input signals from the MCU core: PC, irq, D_{addr}, R_{en}, W_{en}, DMA_{addr} and DMA_{en}. These inputs are used to determine a one-bit *reset* signal output, that, when set to 1, resets the MCU core immediately, i.e., before execution of the next instruction. The *reset* output is triggered when HW-Mod detects any violation of security properties [3].

3.2.4 Verification Pipeline

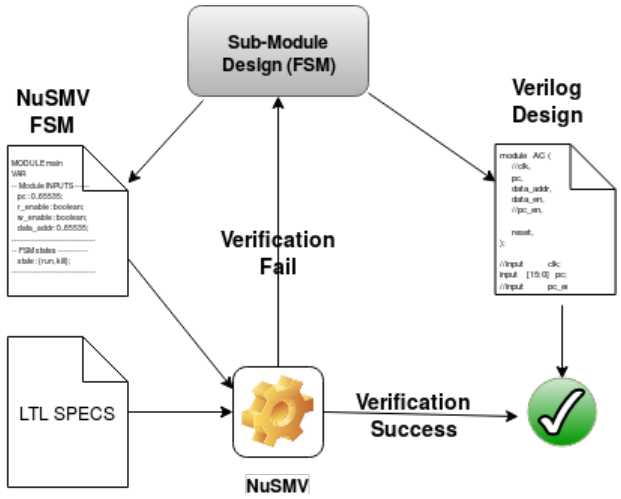


Figure 3.3: *VRASED*'s submodule verification

An overview of HW-Mod verification is shown in Figures 3.3 and 3.4. We start by formalizing RA sub-properties discussed in this section using LTL to define invariants that must hold throughout the entire system execution. HW-Mod is implemented as a composition of sub-

[2] A separate region in RAM is not strictly required. Alternatives and trade-offs are discussed in Section 3.4

[3] Resets due to *VRASED* violations do not give malware advantages as malware can always trigger resets on the unmodified MCU by inducing software faults.

modules written in the Verilog HDL. Each sub-module implements the hardware responsible for ensuring a given subset of the LTL specifications. Each sub-module is described as an FSM in: (1) Verilog at Register Transfer Level (RTL); and (2) the Model-Checking language SMV [34]. We then use the NuSMV model checker to verify that the FSM complies with the LTL specifications. If verification fails, the sub-module is re-designed.

Once each sub-module is verified, they are combined into a single Verilog design. The composition is converted to SMV using the automatic translation tool Verilog2SMV [65]. The resulting SMV is simultaneously verified against all LTL specifications to prove that the final Verilog design for `HW-Mod` complies with all secure RA properties.

We clarify that the individual SMV sub-modules' design and verification steps are not strictly required in the verification pipeline. This is because verifying SMV that is automatically translated from the composition of `HW-Mod` would suffice. Nevertheless, we design FSMs in SMV first so as to facilitate sub-modules' development and reasoning with an early additional check before going into their actual implementation and composition in Verilog.

Remark: *Automatic conversion of the composition of* `HW-Mod` *from Verilog to SMV rules out the possibility of human mistakes in representing Verilog FSMs as SMV.*

For the `SW-Att` part of *VRASED*, we use the HMAC-SHA-256 from the HACL* library [117] to compute an authenticated integrity check of attested memory and $\mathcal{C}$hal received from $\mathcal{V}$rf. This function is formally verified with respect to memory safety, functional correctness, and cryptographic security. However, key secrecy properties (such as clean-up of memory tainted by the key) are not formally verified in HACL* and thus must be ensured by `HW-Mod`.

As the last step, we prove that the conjunction of the LTL properties guaranteed by `HW-Mod` and `SW-Att` implies soundness and security of the RA architecture. These are formally specified in Section 3.3.2.

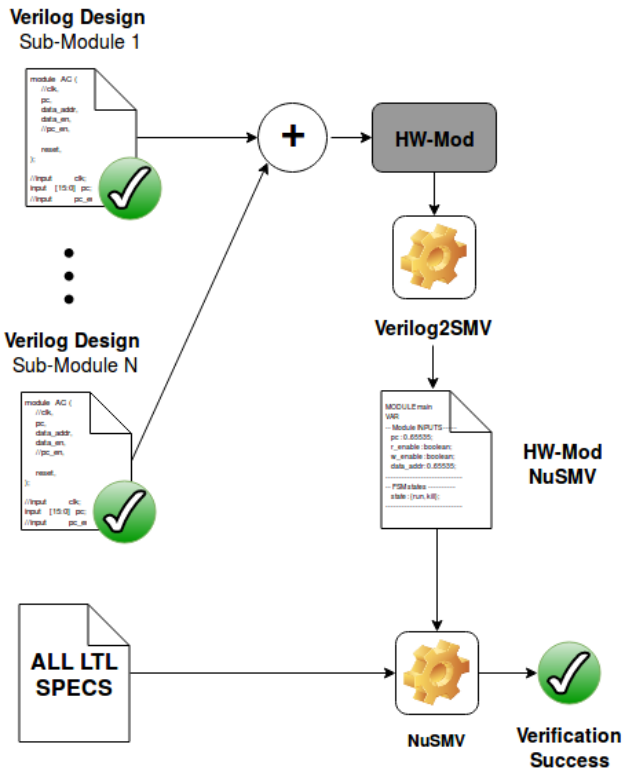


Figure 3.4: Verification framework for the composition of sub-modules (HW-Mod).

3.3 Verifying VRASED

In this section we formalize RA sub-properties. For each sub-property, we represent it as a set of LTL specifications and construct an FSM that is verified to conform to such specifications. Finally, the conjunction of these FSMs is implemented in Verilog HDL and translated to SMV using Verilog2SMV. The generated SMV description for the conjunction is proved to simultaneously hold for all specifications. We also define end-to-end soundness and security goals which are derived from the verified sub-properties (See Appendix 3.8 for the proof).

3.3.1 Notation

To facilitate generic LTL specifications that represent *VRASED*'s architecture (see Figure 3.2) we use the following:

- AR_{min} and AR_{max}: first and last physical addresses of the memory region to be attested;
- CR_{min} and CR_{max}: physical addresses of first and last instructions of `SW-Att` in `ROM`;
- K_{min} and K_{max}: first and last physical addresses of the `ROM` region where $\mathcal{K}$ is stored;
- XS_{min} and XS_{max}: first and last physical addresses of the `RAM` region reserved for `SW-Att` computation;
- MAC_{addr}: fixed address that stores the result of `SW-Att` computation (HMAC);
- MAC_{size}: size of HMAC result in bytes;

Table 3.1 uses the above definitions and summarizes the notation used in our LTL specifications throughout the rest of this chapter.

To simplify specification of defined security properties, we use $[A, B]$ to denote a contiguous memory region between A and B. Therefore, the following equivalence holds:

$$C \in [A, B] \Leftrightarrow (C \leq B \wedge C \geq A) \tag{3.1}$$

For example, expression $PC \in CR$ holds when the current value of PC signal is within CR_{min} and CR_{max}, meaning that the MCU is currently executing an instruction in CR, i.e, a `SW-Att` instruction. This is because in the notation introduced above: $PC \in CR \Leftrightarrow PC \in [CR_{min}, CR_{max}] \Leftrightarrow (PC \leq CR_{max} \wedge PC \geq CR_{min})$.

FSM Representation. As discussed in Section 3.2, `HW-Mod` sub-modules are represented as FSMs that are verified to hold for LTL specifications. These FSMs correspond to the Verilog hardware design of `HW-Mod` sub-modules. The FSMs are implemented as Mealy machines, where output changes at any time as a function of both the current state and current input values[4]. Each FSM has as inputs a subset of the following signals and wires: $\{PC, irq,$

[4]This is in contrast with Moore machines where the output is defined solely based on the current state.

Table 3.1: Summary of *VRASED*-relevant notation

Notation	Description
PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
irq	Signal that indicates if and interrupt is occurring (1-bit)
CR	(Code `ROM`) Memory region where `SW-Att` is stored: $CR = [CR_{min}, CR_{max}]$
KR	($\mathcal{K}$ `ROM`) Memory region where $\mathcal{K}$ is stored: $KR = [K_{min}, K_{max}]$
XS	(eXclusive Stack) secure `RAM` region reserved for `SW-Att` computations: $XS = [XS_{min}, XS_{max}]$
MR	(MAC `RAM`) `RAM` region in which `SW-Att` computation result is written: $MR = [MAC_{addr}, MAC_{addr} + MAC_{size} - 1]$. The same region is also used to pass the attestation challenge as input to `SW-Att`
AR	(Attested Region) Memory region to be attested. Can be fixed/predefined or specified in an authenticated request from $\mathcal{V}$rf: $AR = [AR_{min}, AR_{max}]$
$reset$	A 1-bit signal that reboots the MCU when set to logic 1
A1, A2, ..., A7	Verification axioms (outlined in section 3.2.1)
P1, P2, ..., P7	Properties required for secure RA (outlined in section 3.2.2)

R_{en},W_{en}, D_{addr}, DMA_{en}, DMA_{addr}}.

Each FSM has only one output, *reset*, that indicates whether any security property was violated. For the sake of presentation, we do not explicitly represent the value of the *reset* output for each state. Instead, we define the following implicit representation:

1. *reset* output is 1 whenever an FSM transitions to the *Reset* state;
2. *reset* output remains 1 until a transition leaving the *Reset* state is triggered;
3. *reset* output is 0 in all other states.

3.3.2 Formalizing RA Soundness and Security

We now define the notions of soundness and security. Intuitively, RA soundness corresponds to computing an authenticated integrity ensuring function over a snapshot of attested memory range AR at time some time t. In our case, this function is an HMAC computed on memory AR with a one-time key derived from $\mathcal{K}$ and $\mathcal{C}\mathsf{hal}$. Since `SW-Att` computation is not instantaneous, RA soundness must ensure that attested memory does not change during computation of the HMAC. This is the notion of temporal consistency in remote attestation [29]. In other words, the result of `SW-Att` call must reflect the entire state of the attested memory at the time when `SW-Att` is called. This notion is captured in LTL by Definition 1.

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

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

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

In Definition 1, $PC = CR_{min}$ captures the time when `SW-Att` is called (execution of its first instruction). M and $\mathcal{C}\mathsf{hal}$ are the values of AR and MR at that time. From this precondition, Definition 1 asserts that there is a time in the future when `SW-Att` computation finishes and, at that time, MR stores the result of $HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), M)$. Note that, to satisfy Definition 1, $\mathcal{C}\mathsf{hal}$ and M in the resulting HMAC must correspond to the values in AR and MR, respectively, when `SW-Att` was called.

RA security is defined using the security game in Figure 3.5. It models an adversary $\mathcal{A}\mathsf{dv}$ (as a probabilistic polynomial time machine) that has full control of the software state of $\mathcal{P}\mathsf{rv}$ (as the one described in Section 3.2.1). It can modify AR at will and call `SW-Att` a polynomial number of times in the security parameter (the security parameter corresponds to $\mathcal{K}$ and $\mathcal{C}\mathsf{hal}$ bit-lengths). However, $\mathcal{A}\mathsf{dv}$ can not modify `SW-Att` code, which is stored in immutable memory. The game assumes that $\mathcal{A}\mathsf{dv}$ does not have direct access to $\mathcal{K}$, and only

learns $\mathcal{C}$hal after receiving it from $\mathcal{V}$rf as part of the attestation request.

Definition 2.
2.1 RA ***Security Game (***RA-*game):***
Assumptions:
- *SW-Att is immutable, and $\mathcal{K}$ is not known to $\mathcal{A}$dv*
- *l is the security parameter and $|\mathcal{K}| = |\mathcal{C}\text{hal}| = |MR| = l$*
- *$AR(t)$ denotes the content in AR at time t*
- *$\mathcal{A}$dv can modify AR and MR at will; however, it loses its ability to modify them while SW-Att is running*

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

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

Figure 3.5: RA security definition for *VRASED*

In the following sections, we define SW-Att functional correctness, LTL specifications 3.2-3.10 and formally verify that *VRASED*'s design guarantees such LTL specifications. We derive LTL specifications from the intuitive properties discussed in Section 3.2.2 and depicted in Figure 3.1. In Appendix 3.8 we prove that the conjunction of such properties achieves soundness (Definition 1) and security (Definition 2). For the security proof, we first show that *VRASED* guarantees that $\mathcal{A}$dv can never learn $\mathcal{K}$ with more than negligible probability, thus satisfying the assumption in the security game. We then complete the proof of security via reduction, i.e., show that existence of an adversary that wins the game in Definition 2 implies the existence of an adversary that breaks the conjectured existential unforgeability of HMAC.

Remark: *The rest of this section focuses on conveying the intuition behind the specification of LTL sub-properties. Therefore, our references to the MCU machine model are via Axioms* ***A1 - A7*** *which were described at high level. An LTL machine model formalizing these*

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

Figure 3.6: `SW-Att` C Implementation

notions is described in Appendix 3.8.

3.3.3 *VRASED* `SW-Att`

To minimize required hardware features, hybrid RA approaches implement integrity ensuring functions (e.g., HMAC) in software. *VRASED*'s `SW-Att` implementation is built on top of HACL*'s HMAC implementation [117]. HACL* code is verified to be functionally correct, memory safe and secret independent. In addition, all memory is allocated on the stack making it predictable and deterministic.

`SW-Att` is simple, as depicted in Figure 3.6. It first derives a new unique context-specific key (*key*) from the master key ($\mathcal{K}$) by computing an HMAC-based key derivation function, HKDF [73], on $\mathcal{C}$hal. This key derivation can be extended to incorporate attested memory boundaries if $\mathcal{V}$rf specifies the range (see Appendix 3.9). Finally, it calls HACL*'s HMAC, using *key* as the HMAC key. $ATTEST\_DATA\_ADDR$ and $ATTEST\_SIZE$ specify the memory range to be attested (AR in our notation). We emphasize that `SW-Att` resides in `ROM`, which guarantees **P5** under the assumption of no hardware attacks. Moreover, as discussed below, `HW-Mod` enforces that no other software running on $\mathcal{P}$rv can access memory allocated by `SW-Att` code, e.g., *key*[64] buffer allocated in line 2 of Figure 3.6.

HACL*'s verified HMAC is the core for guaranteeing **P4** (Functional Correctness) in *VRASED*'s design. `SW-Att` functional correctness means that, as long as the memory regions storing values used in `SW-Att` computation (CR, AR, and KR) do not change during its computation,

the result of such computation is the correct HMAC. This guarantee can be formally expressed in LTL as in Definition 3. We note that since HACL*'s HMAC functional correctness is specified in F*, instead of LTL, we manually convert its guarantees to the LTL expressed by Definition 3. By this definition, the value in MR does not need to remain the same, as it will be eventually overwritten by the result of `SW-Att` computation.

Definition 3. *`SW-Att` functional correctness*

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

where M is any arbitrary value for AR.

In addition, some HACL* properties, such as stack-based and deterministic memory allocation, are used in alternative designs of *VRASED* to ensure **P2** – see Section 3.4.

Functional correctness implies that the HMAC implementation conforms to its published standard specification on all possible inputs, retaining the specification's cryptographic security. It also implies that HMAC executes in finite time. Secret independence ensures that there are no branches taken as a function of secrets, i.e., $\mathcal{K}$ and *key* in Figure 3.6. This mitigates leakage of $\mathcal{K}$ (or derived *key*) via timing side-channel attacks. Memory safety guarantees that implemented code is type safe, meaning that it never reads from, or writes to: invalid memory locations, out-of-bounds memory, or unallocated memory. This is particularly important for preventing ROP attacks, as long as **P7** (controlled invocation) is also preserved[5].

Having all memory allocated on the stack allows us to either: (1) confine `SW-Att` execution to a fixed size protected memory region inaccessible to regular software (including malware) running on $\mathcal{P}\mathsf{rv}$; or (2) ensure that `SW-Att` stack is erased before the end of execution. Note that HACL* does not provide stack erasure. Therefore, **P2** does not follow from HACL*

[5]Otherwise, even though the implementation is memory-safe and correct as a whole, chunks of a memory-safe code could still be used in ROP attacks.

implementation. This practice is common because inter-process memory isolation is usually provided by the Operating System (OS). However, erasure before `SW-Att` terminates must be guaranteed. Recall that *VRASED* targets low-end MCUs that might run applications on bare-metal and thus can not rely on any OS features.

As discussed above, even though HACL* implementation guarantees **P4** and storage in `ROM` guarantees **P5**, these must be combined with **P6** and **P7** to provide safe execution. **P6** and **P7** – along with the key protection properties (**P1**, **P2**, and **P3**) – are ensured by `HW-Mod` and are described next.

3.3.4 Key Access Control (`HW-Mod`)

If malware manages to read $\mathcal{K}$ from `ROM`, it can reply to $\mathcal{V}$rf with a forged result. `HW-Mod` access control (AC) sub-module enforces that $\mathcal{K}$ can only be accessed by `SW-Att` (**P1**).

LTL Specification

The invariant for key access control (AC) is defined in LTL Specification (3.2). It stipulates that the system must transition to the *Reset* state whenever code from outside CR tries to read from D_{addr} within the key space.

$$\mathbf{G}: \{\neg(PC \in CR) \wedge R_{en} \wedge (D_{addr} \in KR) \rightarrow reset\} \tag{3.2}$$

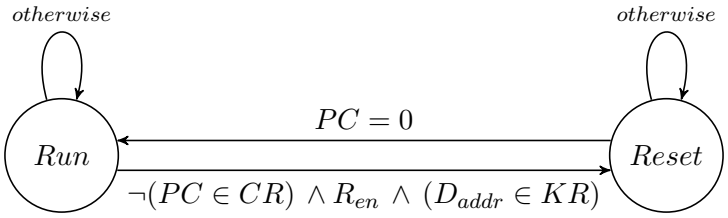


Figure 3.7: Verified FSM for Key AC

Verified Model

Figure 3.7 shows the FSM implemented by the AC sub-module which is verified to hold for LTL Specification 3.2. This FSM has two states: *Run* and *Reset*. It outputs $reset = 1$ when the AC sub-module transitions to state *Reset*. This implies a hard-reset of the MCU. Once the reset process completes, the system leaves the *Reset* state.

3.3.5 Atomicity and Controlled Invocation (`HW-Mod`)

In addition to functional correctness, safe execution of attestation code requires immutability (**P5**), atomicity (**P6**), and controlled invocation (**P7**). **P5** is achieved directly by placing `SW-Att` in `ROM`. Therefore, we only need to formalize invariants for the other two properties: atomicity and controlled execution.

LTL Specification

To guarantee atomic execution and controlled invocation, LTL Specifications (3.3), (3.4) and (3.5) must hold:

$$\mathbf{G}: \{[\neg reset \wedge (PC \in CR) \wedge \neg(\mathbf{X}(PC) \in CR)] \rightarrow [PC = CR_{max} \vee \mathbf{X}(reset)] \} \tag{3.3}$$

$$\mathbf{G}: \{[\neg reset \wedge \neg(PC \in CR) \wedge (\mathbf{X}(PC) \in CR)] \rightarrow [\mathbf{X}(PC) = CR_{min} \vee \mathbf{X}(reset)] \} \quad (3.4)$$

$$\mathbf{G}: \{irq \ \wedge \ (PC \in CR) \ \rightarrow \ reset \} \quad (3.5)$$

LTL Specification (3.3) enforces that the only way for `SW-Att` execution to terminate is through its last instruction: $PC = CR_{max}$. This is specified by checking current and next PC values using LTL ne**X**t operator. In particular, if current PC value is within `SW-Att` region, and next PC value is out of `SW-Att` region, then either current PC value is the address of the last instruction in `SW-Att` (CR_{max}), or $reset$ is triggered in the next cycle. Also, LTL Specification (3.4) enforces that the only way for PC to enter `SW-Att` region is through the very first instruction: CR_{min}. Together, these two invariants imply **P7**: it is impossible to jump into the middle of `SW-Att`, or to leave `SW-Att` before reaching the last instruction.

P6 is satisfied through LTL Specification (3.5). Atomicity could be violated by interrupts. However, LTL Specification (3.5) prevents an interrupt to happen while `SW-Att` is executing. Therefore, if interrupts are not disabled by software running on $\mathcal{P}$rv before calling `SW-Att`, any interrupt that could violate `SW-Att` atomicity will necessarily cause an MCU $reset$.

Verified Model

Figure 3.8 presents a verified model for atomicity and controlled invocation enforcement. The FSM has five states. Two basic states $notCR$ and $midCR$ represent moments when PC points to an address: (1) outside CR, and (2) within CR, respectively, not including

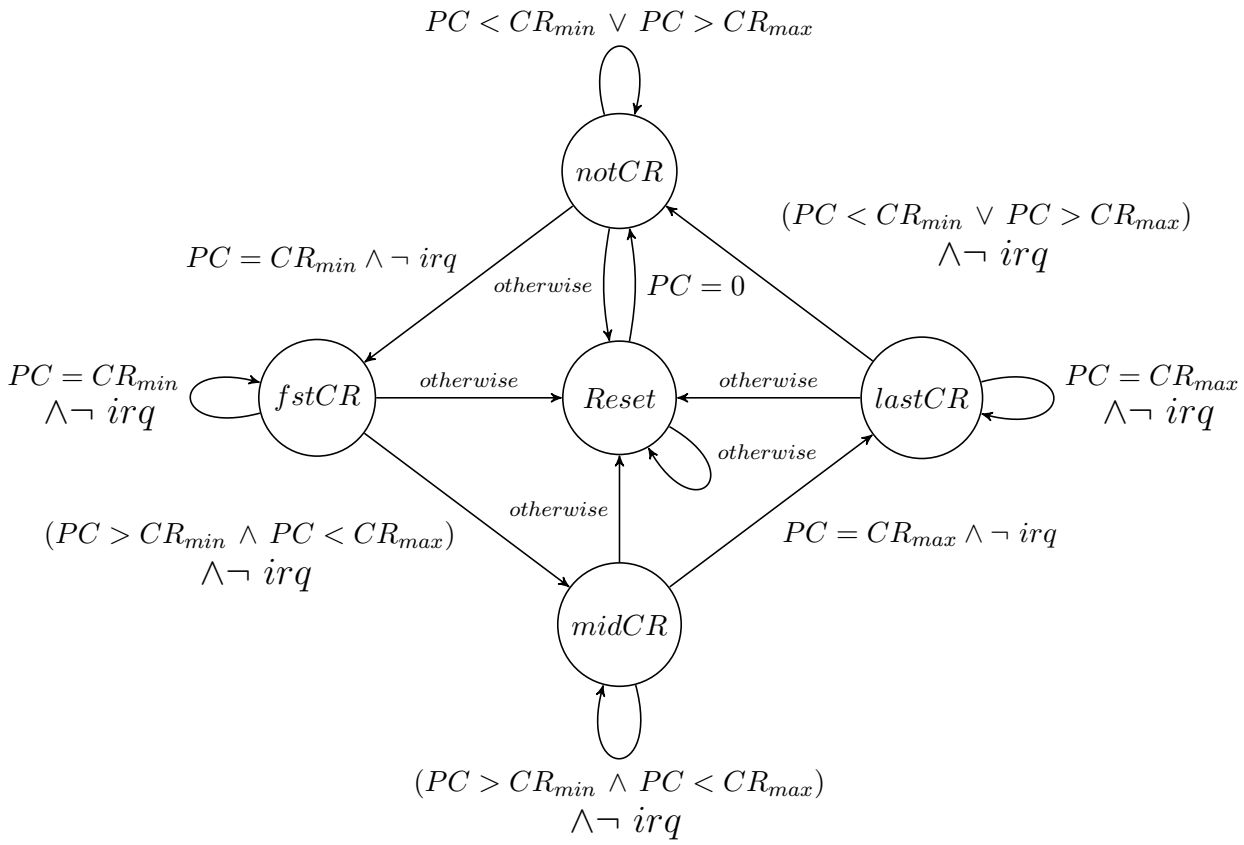


Figure 3.8: Verified FSM for atomicity and controlled invocation.

the first and last instructions of `SW-Att`. Another two: $fstCR$ and $lastCR$ represent states when PC points to the first and last instructions of `SW-Att`, respectively[6]. Note that the only possible path from $notCR$ to $midCR$ is through $fstCR$. Similarly, the only path from $midCR$ to $notCR$ is through $lstCR$. The FSM transitions to the *Reset* state whenever: (1) any sequence of values for PC does not obey the aforementioned conditions; or (2) irq is logical 1 while executing `SW-Att`.

3.3.6 Key Confidentiality (`HW-Mod`)

To guarantee secrecy of $\mathcal{K}$ and thus satisfy **P2**, *VRASED* must enforce the following:

[6]Note that this distinction is possible because `SW-Att` implementation has a single legal entry-point and a single legal exit.

1. No leaks after attestation: any registers and memory accessible to applications must be erased at the end of each attestation instance, i.e., before application execution resumes.
2. No leaks on reset: since a reset can be triggered during attestation execution, any registers and memory accessible to regular applications must be erased upon reset.

Per Axiom **A4**, all registers are zeroed out upon reset and at boot time. Therefore, the only time when register clean-up is necessary is at the end of `SW-Att`. Such clean-up is guaranteed by the Callee-Saves-Register convention: Axiom **A6**.

Nonetheless, the leakage problem remains because of `RAM` allocated by `SW-Att`. Thus, we must guarantee that $\mathcal{K}$ is not leaked through "dead" memory, which could be accessed by application (possibly, malware) after `SW-Att` terminates. A simple and effective way of addressing this issue is by reserving a separate secure stack in `RAM` that is only accessible (i.e., readable and writable) by attestation code. All memory allocations by `SW-Att` must be done on this stack, and access control to the stack must be enforced by `HW-Mod`. As discussed in Section 3.5, the size of this stack is constant – 2.3KBytes. This corresponds to $\approx 3\%$ of MSP430 16-bit address space. We discuss *VRASED* variants that do not require a reserved stack and trade-offs between them in Section 3.4.

LTL Specification

Recall that XS denote a contiguous secure memory region reserved for exclusive access by `SW-Att`. LTL Specification for the secure stack sub-module is as follows:

$$\mathbf{G}: \{\neg(PC \in CR) \wedge (R_{en} \vee W_{en}) \wedge (D_{addr} \in XS) \rightarrow reset\ \} \tag{3.6}$$

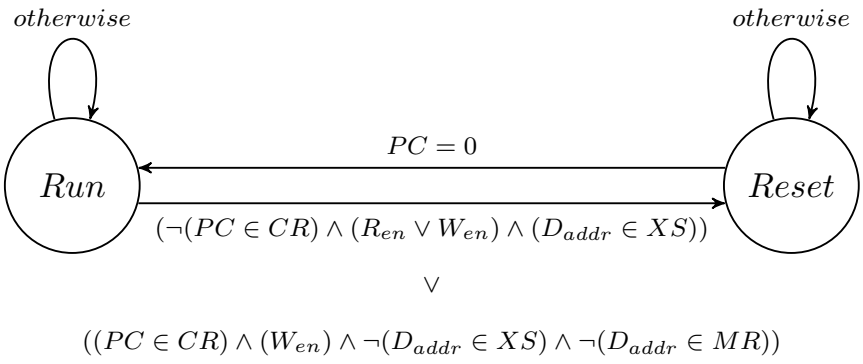


Figure 3.9: Verified FSM for Key Confidentiality

We also want to prevent attestation code from writing into application memory. Therefore, it is only allowed to write to the designated fixed region for the HMAC result (MR).

$$\mathbf{G}: \{(PC \in CR) \wedge (W_{en}) \wedge \neg(D_{addr} \in XS) \wedge \neg(D_{addr} \in MR) \rightarrow reset\} \tag{3.7}$$

In summary, invariants (3.6) and (3.7) enforce that only attestation code can read from/write to the secure reserved stack and that attestation code can only write to regular memory within the space reserved for the HMAC result. If any of these conditions is violated, the system resets.

Verified Model

Figure 3.9 shows the FSM verified to comply with invariants (3.6) and (3.7).

3.3.7 DMA Support

So far, we presented a formalization of `HW-Mod` sub-modules under the assumption that DMA is either not present or disabled on $\mathcal{P}$`rv`. However, when present, a DMA controller can access arbitrary memory regions. Such memory access is performed concurrently in the memory

backbone and without MCU intervention, while the MCU executes regular instructions.

DMA data transfer is performed using dedicated memory buses, e.g., DMA_{en} and DMA_{addr}. Hence, regular memory access control (based on monitoring D_{addr}) does not apply to memory access by DMA controller. Thus, if DMA controller is compromised, it may lead to violation of **P1** and **P2** by directly reading $\mathcal{K}$ and values in the attestation stack, respectively. In addition, it can assist $\mathcal{P}$rv-resident malware to escape detection by either copying it out of the measurement range or deleting it, which results in a violation of **P6**.

LTL Specification

We introduce three additional LTL Specifications to protect against aforementioned attacks. First, we enforce that DMA cannot access $\mathcal{K}$.

$$\mathbf{G}: \{DMA_{en} \wedge (DMA_{addr} \in KR) \rightarrow reset\} \tag{3.8}$$

Similarly, LTL Specification for preventing DMA access to the attestation stack is defined as:

$$\mathbf{G}: \{DMA_{en} \wedge (DMA_{addr} \in XS) \rightarrow reset\} \tag{3.9}$$

Finally, invariant (3.10) specifies that DMA must be always disabled while PC is in `SW-Att` region. This prevents DMA controller from helping malware escape during attestation.

$$\mathbf{G}: \{(PC \in CR) \wedge DMA_{en} \rightarrow reset\} \tag{3.10}$$

Verified Model

Figure 3.10 shows the FSM verified to comply with invariants (3.8) to (3.10).

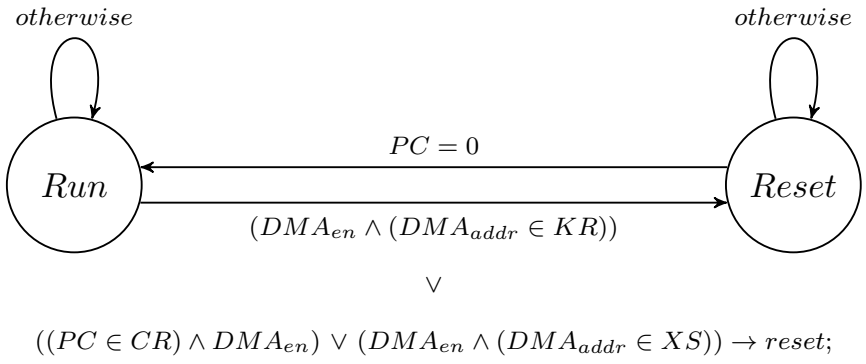


Figure 3.10: Verified FSM for DMA protection

3.3.8 `HW-Mod` Composition

Thus far, we designed and verified individual `HW-Mod` sub-modules according to the methodology in Section 3.2.4 and illustrated in Figure 3.3. We now follow the workflow of Figure 3.4 to combine the sub-modules into a single Verilog module. Since each sub-module individually guarantees a subset of properties **P1**–**P7**, the composition is simple: the system must reset whenever any sub-module reset is triggered. This is implemented by a logical OR of sub-modules reset signals. The composition is shown in Figure 3.11.

To verify that all LTL specifications still hold for the composition, we use Verilog2SMV [65] to translate `HW-Mod` to SMV and verify SMV for all of these specifications simultaneously.

3.3.9 Secure Reset (`HW-Mod`)

Finally, we define an LTL Specification for secure reset (**P3**). According to Axiom **A4**, all registers (including PC) are set to 0 on reset. However, the reset routine implemented by

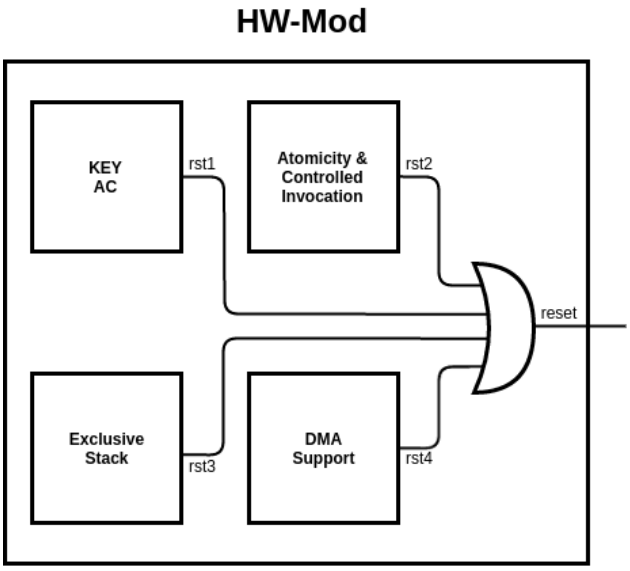


Figure 3.11: `HW-Mod` composition from sub-modules

the MCU might take several clock cycles. Ensuring that $reset = 1$ until the point when registers are wiped is important in order to guarantee that $\mathcal{K}$ is not leaked through registers after a reset. That is because some part of $\mathcal{K}$ might remain in some of the registers if a reset happens during `SW-Att` execution.

LTL Specification

To guarantee that the reset signal is active for long enough so that the MCU reset finishes and all registers are cleaned-up, it must hold that:

$$\mathbf{G}: \{reset \rightarrow [(reset \quad \mathbf{U} \quad PC = 0) \quad \vee \quad \mathbf{G}(reset)] \} \tag{3.11}$$

Invariant (3.11) states: when reset signal is triggered, it can only be released after $PC = 0$. Transition from *Reset* state in all sub-modules presented in this section already takes this invariant into account. Thus, `HW-Mod` composition also verifies LTL Specification (3.11).

3.4 Alternative Designs

We now discuss alternative designs for *VRASED* that guarantee verified properties without requiring a separate secure stack region for SW-Att operations. Recall that HW-Mod enforces that only SW-Att can access this stack. Since memory usage in HACL* HMAC is deterministic, the size of the separate stack can be pre-determined – $2,332$bytes, in this specific implementation. Even though resulting in overall (HW and SW) design simplicity, dedicating 3% of addressable memory to secure RA might not be desirable. Therefore, we consider several alternatives. In Section 3.5 the costs involved with these alternatives are quantified and compared to the standard design of *VRASED*.

3.4.1 Erasure on SW-Att

The most intuitive alternative to a reserved secure stack (which prevents accidental key leakage by SW-Att) is to encode corresponding properties into the HACL* implementation and proof. Specifically, it would require extending the HACL* implementation to zero out all allocated memory before every function return. In addition, to retain verification of **P2** (in Section 3.2.2) and ensure no leakage, HACL*-verified properties must be extended to incorporate memory erasure. This is not yet supported in HACL* and doing so would incur a slight performance overhead. However, the trade-off between performance and RAM savings might be worthwhile.

At the same time, we note that, even with verified erasure as a part of SW-Att, **P2** is still not guaranteed if the MCU does not guarantee erasure of the entire RAM upon boot. This is necessary in order to consider the case when $\mathcal{P}$rv re-boots in the middle of SW-Att execution. Without a reserved stack, $\mathcal{K}$ might persist in RAM. Since the memory range for SW-Att execution is not fixed, hardware support is required to bootstrap secure RAM erasure

before starting any software execution. In fact, such support is necessary for all approaches without a separate secure stack.

3.4.2 Compiler-Based Clean-Up

While stack erasure in HACL* would integrate nicely with the overall proof of `SW-Att`, the assurance would be at the language abstraction level, and not necessarily at the machine level. The latter would require additional assumptions about the compilation tool chain. We could also consider performing stack erasure directly in the compiler. In fact, a recent proposal to do exactly that was made in zerostack [103], an extension to Clang/LLVM. In case of *VRASED*, this feature could be used on unmodified HACL* (at compilation time), to add instructions to erase the stack before the return of each function enabling **P2**, assuming the existence of a verified `RAM` erasure routine upon boot. We emphasize that this approach may increase the compiler's trusted code base. Ideally, it should be implemented and formally verified as part of a verified compiler suite, such as CompCert [74].

3.4.3 Double-HMAC Call

Finally, complete stack erasure could also be achieved directly using currently verified HACL* properties, without any further modifications. This approach involves invoking HACL* HMAC function a second time, after the computation of the actual HMAC. The second "dummy" call would use the same input data, however, instead of using $\mathcal{K}$, an independent constant, such as $\{0\}^{512}$, would be used as the HMAC key.

Recall that HACL* is verified to only allocate memory on the stack in a deterministic manner. Also, due to HACL*'s verified properties that mitigate side-channels, software flow does not change based on the secret key. Therefore, this deterministic allocation implies that, for

inputs of the same size, any variable allocated by the first "real" HMAC call (tainted by $\mathcal{K}$), would be overwritten by the corresponding variable in the second "dummy" call. Note that the same guarantee discussed in Section 3.4.1 is provided here and secure `RAM` erasure at boot would still be needed for the same reasons. Admittedly, this double-HMAC approach would consume twice as many CPU cycles. Still, it might be a worthwhile trade-off, especially, if there is memory shortage and lack of previously discussed HACL* extension or compiler extension.

3.5 Evaluation

We now discuss implementation details and evaluate *VRASED*'s overhead and performance. Section 3.5.2 reports on verification complexity. Section 3.5.3 discusses performance in terms of time and space complexity as well as its hardware overhead. We also provide a comparison between *VRASED* and other RA architectures targeting low-end devices, namely SANCUS [87] and SMART [49], in Section 3.5.4.

3.5.1 Implementation

As mentioned earlier, we use OpenMSP430 [56] as an open core implementation of the MSP430 architecture. OpenMSP430 is written in the Verilog HDL and can execute software generated by any MSP430 toolchain with near cycle accuracy. We modified the standard OpenMSP430 to implement the hardware architecture presented in Section 3.2.3, as shown in Figure 3.2. This includes adding `ROM` to store $\mathcal{K}$ and `SW-Att`, adding `HW-Mod`, and adapting the memory backbone accordingly. We use Xilinx Vivado [113] – a popular logic synthesis tool – to synthesize an RTL description of `HW-Mod` into hardware in FPGA. FPGA synthesized hardware consists of a number of logic cells. Each consists of Look-Up Tables (LUTs) and

registers; LUTs are used to implement combinatorial boolean logic while registers are used for sequential logic elements, i.e., FSM states and data storage. We compiled `SW-Att` using the native msp430-gcc [107] and used Linker scripts to generate software images compatible with the memory layout of Figure 3.2. Finally, we evaluated *VRASED* on the FPGA platform Artix-7 [114].

3.5.2 Verification Results

As discussed in Section 3.2.2, *VRASED*'s verification consists of properties **P1**–**P7**. **P5** is achieved directly by executing `SW-Att` from `ROM`. Meanwhile, HACL* HMAC verification implies **P4**. All other properties are automatically verified using NuSMV model checker. Table 3.2 shows the verification results of *VRASED*'s `HW-Mod` composition as well as results for individual sub-modules. It shows that *VRASED* successfully achieves all the required security properties. These results also demonstrate feasibility of our verification approach, since the verification process – running on a commodity desktop computer – consumes only small amount of memory and time: $<$ 14MB and 0.3sec, respectively, for all properties.

Table 3.2: Verification results running on a desktop @ 3.40 GHz.

HW Submod.	LTL Spec.	Mem. (MB)	Time (s)	Verified
Key AC	3.2,3.11	7.5	.02	✓
Atomicity	3.3,3.4,3.5,3.11	8.5	.05	✓
Exclusive Stack	3.6,3.7,3.11	8.1	.03	✓
DMA Support	3.8-3.11	8.2	.04	✓
`HW-Mod`	3.2-3.11	13.6	.28	✓

Table 3.3: Evaluation of cost, overhead, and performance of RA

Method	RAM Erasure Required Upon Boot?	FPGA Hardware LUT	Reg	Cell	Verilog LoC	Memory (byte) ROM	Sec. RAM	Time to attest 4KB CPU cycles	ms (at 8MHz)
Core (Baseline)	N/A	1842	684	3044	4034	0	0	N/A	N/A
Secure Stack (Section 3.3)	No	1964	721	3237	4621	4500	2332	3601216	450.15
Erasure on SW-Att (Section 3.4.1)	Yes	1954	717	3220	4516	4522	0	3613283	451.66
Compiler-based Clean-up (Section 3.4.2) [7]	Yes	1954	717	3220	4516	4522	0	3613283	451.66
Double-HMAC Call (Section 3.4.3)	Yes	1954	717	3220	4516	4570	0	7201605	900.20

Table 3.4: Qualitative comparison between RA architectures targeting low-end devices

	VRASED	SMART	SANCUS
Design Type	Hybrid (HW/SW)	Hybrid (HW/SW)	Pure HW
RA function	HMAC-SHA256	HMAC-SHA1	SPONGENT-128/128/8
ROM for RA code	Yes	Yes	No
DMA Support	Yes	No	No
Formally Verified	Yes	No	No

3.5.3 Performance and Hardware Cost

We now report on *VRASED*'s performance considering the standard design (described in Section 3.3) and alternatives discussed in Section 3.4. We evaluate the hardware footprint, memory (ROM and secure RAM), and run-time. Table 3.3 summarizes the results.

Hardware Footprint. *VRASED*'s standard approach (with the exclusive stack) adds around 587 lines of code in Verilog HDL. This corresponds to around 15% of the code in the original OpenMSP430 core. In terms of synthesized hardware, it requires 122 (6.6%) and 37 (5.4%) additional LUTs and registers, respectively. Overall, *VRASED* contains 193 logic cells more than the unmodified OpenMSP430 core, corresponding to a 6.3% increase.

Memory. *VRASED* requires ∼4.5KB of ROM; most of which (96%) is for storing HACL* HMAC-SHA256 code. The secure stack approach has the smallest ROM size, as it does not need to perform a memory clean-up in software. However, this advantage is attained at the price of requiring 2.3KBytes of reserved RAM. This overhead corresponds to 3.5% of MSP430 16-bit address space.

Attestation Run-time. Attestation run-time is dominated by the time it takes to compute the HMAC of $\mathcal{P}$rv's memory. The secure stack, erasure on SW-Att and compiler-based

[7] As mentioned in Section 3.4.2, there is no formally verified msp430 compiler capable of performing stack erasure. Thus, we estimate overhead of this approach by manually inserting code required for erasing the stack in SW-Att.

clean-up approaches take roughly $.45s$ to attest $4KB$ of RAM on an MSP430 device with a clock frequency at 8MHz. Whereas, the double MAC approach requires invoking the HMAC function twice, leading its run-time to roughly double.

Discussion. We consider *VRASED*'s overhead to be affordable. The additional hardware, including registers, logic gates and exclusive memory, resulted in only a 3-6% increase. The number of cycles required by SW-Att exhibits a linear increase with the size of attested memory. As MSP430 typically runs at 8-25MHz, attestation of the entire RAM on a typical MSP430 can be computed in less than a second. *VRASED*'s RA is relatively cheap to the $\mathcal{P}$rv. As a point of comparison we can consider a common cryptographic primitive such as the Curve25519 Elliptic-Curve Diffie-Hellman (ECDH) key exchange. A single execution of an optimized version of such protocol on MSP430 has been reported to take $\approx$ 9 million cycles [59]. As Table 3.3 shows, attestation of 4KBytes (typical size of RAM in some MSP430 models) can be computed three times faster.

3.5.4 Comparison with Other Low-End RA Architectures

We here compare *VRASED*'s overhead with two previous RA architectures targeting low-end embedded systems: SMART [49] and SANCUS [87]. We emphasize, however, that both SMART and SANCUS were designed in an ad hoc manner. Thus, they can not be formally verified and do not provide any guarantees offered by *VRASED*'s verified architecture. Nevertheless, we contrast *VRASED*'s cost with such architectures to demonstrate its affordability.

Table 3.4 presents a comparison between features offered and required by aforementioned architectures. SANCUS is, to the best of our knowledge, the cheapest pure HW-based architecture, while SMART is a minimal HW/SW RA co-design. Since SANCUS's RA routine is implemented entirely in HW, it does not require ROM to store the SW implementation of

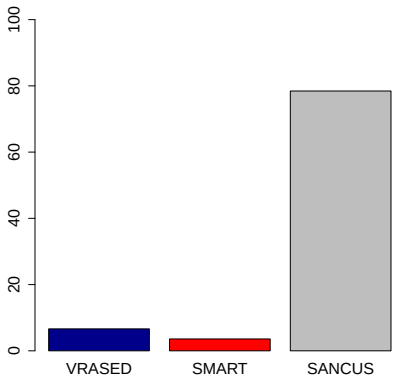
the integrity ensuring function. *VRASED* implements a MAC with digest sizes of 256-bits. SMART and SANCUS, on the other hand, use SHA1-based MAC and SPONGENT-128/128/8 [20], respectively. Such MACs do not offer strong collision resistance due to the small digest sizes (and known collisions). Of the three architectures, *VRASED* is the only one secure in the presence of DMA and the only one to be rigorously specified and formally verified.

Figure 3.12 presents a quantitative comparison between the RA architectures. It considers additional overhead in relation to the latest version of the unmodified OpenMSP430 (available at [56]). Compared to *VRASED*, SANCUS requires 12× more Look-Up Tables, 22× more registers, and its (unverified) TCB is 2.5 times larger in lines of Verilog code. This comparison demonstrates the cost of relying on a HW-only approach even when designed for minimality. SMART's overhead is slightly smaller than that of *VRASED* due to lack of DMA support. In terms of attestation execution time, SMART is the slowest, requiring 9.2M clock cycles to attest 4KB of memory. SANCUS achieves the fastest attestation time (1.3M cycles) due to the HW implementation of SPONGENT-128/128/8. *VRASED* sits in between the two with a total attestation time of 3.6M cycles.

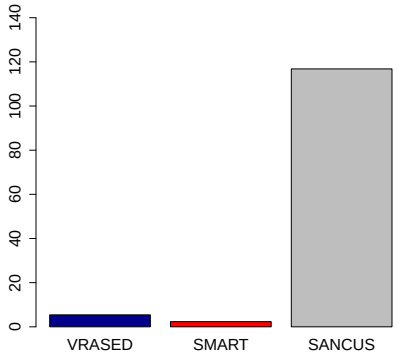
3.6 Related Work

We are unaware of any previous work that yielded a formally verified RA design. Nevertheless, formal verification has been widely used as the *de facto* means to guarantee that a system is free of implementation errors and bugs. In recent years, several efforts focused on verifying security-critical systems.

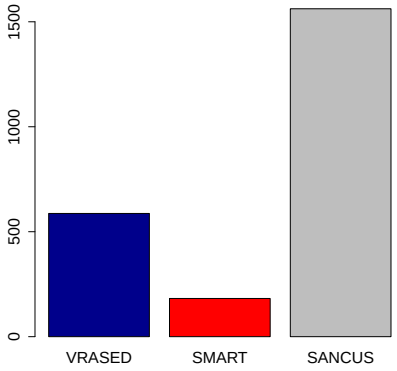
In terms of cryptographic primitives, Hawblitzel et al. [58] verified new implementations of SHA, HMAC, and RSA. Beringer et al.[17] verified the Open-SSL SHA-256 implemen-



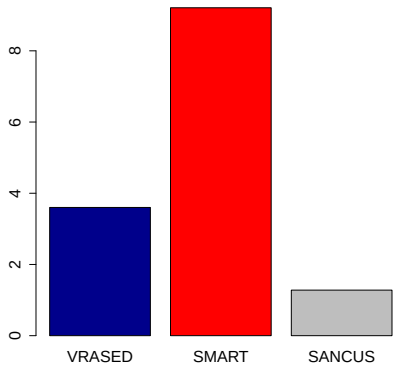
(a) Additional HW overhead (%) in Number of Look-Up Tables



(b) Additional HW overhead (%) in Number of Registers



(c) Additional Verilog Lines of Code



(d) Time to attest 4KB (in millions of CPU cycles)

Figure 3.12: Comparison between RA architectures targeting low-end devices

tation. Bond et al. [21] verified an assembly implementation of SHA-256, Poly1305, AES and ECDSA. More recently, Zinzindohoué, et al. [117] developed HACL*, a verified cryptographic library containing the entire cryptographic API of NaCl [18]. As discussed earlier, HACL*'s verified HMAC forms the core of *VRASED*'s software component.

Larger security-critical systems have also been successfully verified. For example, Bhargavan [19] implemented the TLS protocol with verified cryptographic security. CompCert[74] is a C compiler that is formally verified to preserve C code semantics in generated assembly code. Klein et al. [67] designed and proved functional correctness of `seL4` – the first verified

general-purpose microkernel. More recently, Tuncay et al. verified a design for Android OS App permissions model [110].

The importance of verifying RA has been recently acknowledged by Lugou et al. [80], which discussed methodologies for specifically verifying HW/SW RA co-designs. A follow-on result proposed the SMASH-UP tool [81]. By modeling a hardware abstraction, SMASH-UP allows automatic conversion of assembly instructions to the effects on hardware representation. Similarly, Cabodi et al. [26, 25] discussed the first steps towards formalizing hybrid RA properties. However, none of these results yielded a fully verified (and publicly available) RA architecture, such as *VRASED*.

3.7 Conclusion

This chapter presented *VRASED* – the first formally verified hybrid RA method that uses a verified cryptographic software implementation and combines it with a verified hardware design to guarantee correct implementation of RA security properties. *VRASED* is also the first verified security service implemented as a HW/SW co-design. *VRASED* was designed with simplicity and minimality in mind. It results in efficient computation and low hardware cost, realistic even for low-end embedded systems. *VRASED*'s practicality is demonstrated via publicly available implementation using the low-end MSP430 platform.

APPENDIX

3.8 Appendix: RA Soundness and Security Proofs

3.8.1 Proof Strategy

We present the proofs for RA soundness (Definition 1) and RA security (Definition 2). Soundness is proved entirely via LTL equivalences. In the proof of security we first show, via LTL equivalences, that *VRASED* guarantees that adversary $\mathcal{A}\mathsf{dv}$ can never learn $\mathcal{K}$ with more than negligible probability. We then prove security by showing a reduction of HMAC's existential unforgeability to *VRASED*'s security. In other words, we show that existence of $\mathcal{A}\mathsf{dv}$ that breaks *VRASED* implies existence of HMAC-$\mathcal{A}\mathsf{dv}$ able to break conjectured existential unforgeability of HMAC. The full machine-checked proofs for the LTL equivalences (using Spot 2.0 [47] proof assistant) discussed in the remainder of this section are available in [1].

3.8.2 Machine Model

To prove that *VRASED*'s design satisfies end-to-end definitions of soundness and security for RA, we start by formally defining (in LTL) memory and execution models corresponding to the architecture introduced in Section 3.2.

Definition 4 (Memory model)**.**

1. *$\mathcal{K}$ is stored in* `ROM` $\leftrightarrow G : \{KR = \mathcal{K}\}$
2. *`SW-Att` is stored in* `ROM` $\leftrightarrow G : \{CR =$ `SW-Att`$\}$
3. *MR, CR, AR, KR, and XS are non-overlapping memory regions*

The memory model in Definition 4 captures that KR and CR are `ROM` regions, and are thus immutable. Hence, the values stored in those regions always correspond to $\mathcal{K}$ and `SW-Att` code, respectively. Finally, the memory model states that MR, CR, AR, KR, and XS are disjoint regions in the memory layout, corresponding to the architecture in Figure 3.2.

Definition 5 (Execution model)**.**

1. *Modify_Mem(i)* $\rightarrow (W_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$
2. *Read_Mem(i)* $\rightarrow \ (R_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$
3. *Interrupt* $\rightarrow irq$

Our execution model, in Definition 5, translates MSP430 behavior by capturing the effects on the processor signals when reading and writing from/to memory. We do not model the effects of instructions that only modify register values (e.g., ALU operations, such as *add* and *mul*) because they are not necessary in our proofs.

The execution model defines that a given memory address can be modified in two cases: by a CPU instruction or by DMA. In the first case, the W_{en} signal must be on and D_{addr} must contain the memory address being accessed. In the second case, DMA_{en} signal must be on and DMA_{addr} must contain the address being modified by DMA. The requirements for reading from a given address are similar, except that instead of W_{en}, R_{en} must be on. Finally, the execution model also captures the fact that an interrupt implies setting the irq signal to 1.

3.8.3 RA Soundness Proof

The proof follows from `SW-Att` functional correctness (expressed by Definition 3) and LTL specifications 3.3, 3.5, 3.7, and 3.10

Theorem 1. *VRASED is sound according to Definition 1.*

Proof.

$$Definition\ 3\ \wedge\ LTL_{3.3} \wedge LTL_{3.5} \wedge LTL_{3.7} \wedge LTL_{3.10} \rightarrow Theorem\ 1$$

□

The formal computer proof for Theorem 1 can be found in [1]. The intuition for this proof is two-part. First, `SW-Att` functional correctness (Definition 3) would imply Theorem 1 if AR, CR, KR never change and an interrupt does not happen during `SW-Att` computation. However, memory model Definitions 4.1 and 4.2 already guarantee that CR and KR never change. Also, LTL 3.5 states that an interrupt cannot happen during `SW-Att` computation, otherwise the device resets. Therefore, it remains for us to show that AR does not change during `SW-Att` computation. This is stated in Lemma 1.

Lemma 1. *Temporal Consistency – Attested memory does not change during* `SW-Att` *computation*

$$\boldsymbol{G}: \{$$
$$PC = CR_{min} \wedge AR = M \wedge \neg reset\ \boldsymbol{U}\ (PC = CR_{max}) \rightarrow$$
$$(AR = M)\ \boldsymbol{U}\ (PC = CR_{max})\ \}$$

In turn, Lemma 1 can be proved by:

$$LTL_{3.3} \wedge LTL_{3.7} \wedge LTL_{3.10} \rightarrow Lemma\ 1 \tag{3.12}$$

The reasoning for Equation 3.12 is as follows:

- $LTL_{3.3}$ prevents the CPU from stopping execution of `SW-Att` before its last instruction.

- $LTL_{3.7}$ guarantees that the only memory regions written by the CPU during `SW-Att` execution are XS and MR, which do not overlap with AR.
- $LTL_{3.10}$ prevents DMA from writing to memory during `SW-Att` execution.

Therefore, there are no means for modifying AR during `SW-Att` execution, implying Lemma 1. As discussed above, it is easy to see that:

$$Lemma\ 1 \wedge LTL_{3.5} \wedge Definition\ 3 \rightarrow Theorem\ 1 \tag{3.13}$$

3.8.4 RA Security Proof

Recall the definition of RA security in the game in Figure 3.5. The game makes two key assumptions:

1. `SW-Att` call results in a temporally consistent HMAC of AR using a key derived from $\mathcal{K}$ and $\mathcal{C}$hal. This is already proved by *VRASED*'s soundness.
2. $\mathcal{A}$dv never learns $\mathcal{K}$ with more than negligible probability.

Lemma 2. *Key confidentiality – $\mathcal{K}$ can not be accessed directly by untrusted software ($\neg(PC \in CR)$) and any memory written to by `SW-Att` can never be read by untrusted software.*

$$\begin{aligned}
&\boldsymbol{G}: \{ \\
&(\neg(PC \in CR) \wedge Read_Mem(i) \wedge i \in KR \rightarrow reset) \wedge \\
&(DMA_{en} \wedge DMA_{addr} = i \wedge i \in KR \rightarrow reset) \wedge \\
&[\neg reset \wedge PC \in CR \wedge Modify_Mem(i) \wedge \neg(i \in MR) \rightarrow \\
&\boldsymbol{G}: \{(\neg(PC \in CR) \wedge Read_Mem(i) \vee DMA_{en} \wedge DMA_{addr} = i) \\
&\rightarrow reset\}] \\
&\}
\end{aligned}$$

By proving that *VRASED*'s design satisfies assumptions 1 and 2, we show that the capabilities of untrusted software (any DMA or CPU software other than `SW-Att`) on $\mathcal{P}$rv are

equivalent to the capabilities of $\mathcal{A}\mathsf{dv}$ in RA-game. Therefore, we still need to prove item 2 before we can use such game to prove *VRASED*'s security. The proof of $\mathcal{A}\mathsf{dv}$'s inability to learn $\mathcal{K}$ with more than negligible probability is facilitated by *A6 - Callee-Saves-Register* convention stated in Section 3.2. A6 directly implies no leakage of information through registers on the return of `SW-Att`. This is because, before the return of a function, registers must be restored to their state prior to the function call. Thus, untrusted software can only learn $\mathcal{K}$ (or any function of $\mathcal{K}$) through memory. However, if untrusted software can never read memory written by `SW-Att`, it never learns anything about $\mathcal{K}$ (the secret-independence of `SW-Att` at the HACL* level even implies a lack of timing side-channels, subject to our assumption that this property is preserved by msp430-gcc and the MCU implementation). Now, it suffices to prove that untrusted software can not access $\mathcal{K}$ directly and that it can never read memory written by `SW-Att`. These conditions are stated in LTL in Lemma 2. We prove that *VRASED* satisfies Lemma 2 by writing a computer proof (available in [1]) for Equation 3.14. The reasoning for this proof is similar to that of RA soundness and omitted.

$$LTL_{3.2} \wedge LTL_{3.6} \wedge LTL_{3.7} \wedge LTL_{3.8} \wedge LTL_{3.9} \wedge LTL_{3.10} \rightarrow \text{Lemma 2} \tag{3.14}$$

We emphasize that Lemma 2 does not restrict reads and writes to MR, since this memory is used for inputting $\mathcal{C}\mathsf{hal}$ and receiving `SW-Att` result. Nonetheless, the already proved RA soundness and LTL 3.4 (which makes it impossible to execute fractions of `SW-Att`) guarantee that MR will not leak anything, because at the end of `SW-Att` computation it will always contain an HMAC result, which does not leak information about $\mathcal{K}$. After proving Lemma 2, the capabilities of untrusted software on $\mathcal{P}\mathsf{rv}$ are equivalent to those of adversary $\mathcal{A}\mathsf{dv}$ in RA-game of Definition 2. Therefore, in order to prove *VRASED*'s security, it remains to show a reduction from HMAC security according to the game in Definition 2. *VRASED*'s security is stated and proved in Theorem 2.

Theorem 2. *VRASED is secure according to Definition 2 as long as HMAC is a secure* MAC.

Proof. A MAC *is defined as tuple of algorithms* $\{\texttt{Gen}, \texttt{Mac}, \texttt{Vrf}\}$. *For the reduction we construct a slightly modified HMAC', which has the same* `Mac` *and* `Vrf` *algorithms as standard HMAC but* $\texttt{Gen} \leftarrow KDF(\mathcal{K}, \mathcal{C}\mathsf{hal})$ *where* $\mathcal{C}\mathsf{hal} \leftarrow \$\{0,1\}^l$. *Since KDF function itself is implemented as a* `Mac` *call, it is easy to see that the outputs of* `Gen` *are indistinguishable from random. In other words, the security of this slightly modified construction follows from the security of HMAC itself. Assuming that there exists* $\mathcal{A}\mathsf{dv}$ *such that* $Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA}_{game}] > negl(l)$, *we show that such adversary can be used to construct HMAC-*$\mathcal{A}\mathsf{dv}$ *that breaks existential unforgeability of HMAC' with probability* $Pr[HMAC\text{-}\mathcal{A}\mathsf{dv}\ , \mathsf{MAC}\text{-}game] > negl(l)$. *To that purpose HMAC-*$\mathcal{A}\mathsf{dv}$ *behaves as follows:*

1. *HMAC-*$\mathcal{A}\mathsf{dv}$ *selects msg to be the same* $M \neq AR$ *as in* RA*-game and asks* $\mathcal{A}\mathsf{dv}$ *to produce the same output used to win* RA*-game.*
2. *HMAC-*$\mathcal{A}\mathsf{dv}$ *outputs the pair (msg,*σ*) as a response for the challenge in the standard existential unforgeability game, where* σ *is the output produced by* $\mathcal{A}\mathsf{dv}$ *in step 1.*

By construction, (msg,σ*) is a valid response to a challenge in the existential unforgeability* MAC *game considering HMAC' as defined above. Therefore, HMAC-*$\mathcal{A}\mathsf{dv}$ *is able to win the existential unforgeability game with the same* $> negl(l)$ *probability that* $\mathcal{A}\mathsf{dv}$ *has of winning* RA*-game in Definition 2.* □

3.9 Appendix: Verifier Authentication

Depending on the setting where $\mathcal{P}\mathsf{rv}$ is deployed, authenticating the attestation request before executing `SW-Att` may be required. For example, if $\mathcal{P}\mathsf{rv}$ is in a public network, the

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

Figure 3.13: `SW-Att` Implementation with $\mathcal{V}$rf authentication

adversary may try to communicate with it. In particular, the adversary can impersonate $\mathcal{V}$rf and send fake attestation requests to $\mathcal{P}$rv, attempting to cause denial-of-service. This is particularly relevant if $\mathcal{P}$rv is a safety-critical device. If $\mathcal{P}$rv receives too many attestation requests, regular (and likely honest) software running on $\mathcal{P}$rv would not execute because `SW-Att` would run all the time. Thus, we now discuss an optional part of *VRASED*'s design suitable for such settings. It supports authentication of $\mathcal{V}$rf as part of `SW-Att` execution. Our implementation is based on the protocol in [50], where the attestation challenge becomes a monotonically increasing counter.

Figure 3.13 presents an implementation of `SW-Att` that includes $\mathcal{V}$rf authentication. It also builds upon HACL* verified HMAC to authenticate $\mathcal{V}$rf, in addition to computing the authenticated integrity check. In this case, $\mathcal{V}$rf's request contains an HMAC of the challenge computed using $\mathcal{K}$. Before calling `SW-Att`, software running on $\mathcal{P}$rv is expected to store the received challenge on a fixed address $CHALL\_ADDR$ and the corresponding received HMAC on VRF_AUTH. `SW-Att` discards the attestation request if (1) the received challenge is less than or equal to the latest challenge, or (2) HMAC of the received challenge is mismatched. After that, it derives a new unique key using HKDF [73] from $\mathcal{K}$ and the

received HMAC and uses it as the attestation key.

`HW-Mod` must also be slightly modified to ensure security of $\mathcal{V}$rf's authentication. In particular, regular software must not be able modify the memory region that stores $\mathcal{P}$rv's counter. Notably, the counter requires persistent and writable storage, because `SW-Att` needs to modify it at the end of each attestation execution. Therefore, CTR region resides on FLASH (FLASH is persistent). We denote this region as:

- $CTR = [CTR_{min}, CTR_{max}]$;

LTL Specifications (3.15) and (3.16) must hold (in addition to the ones discussed in Section 3.3).

$$\mathbf{G}: \{\neg(PC \in CR) \wedge W_{en} \wedge (D_{addr} \in CTR) \rightarrow reset\} \tag{3.15}$$

$$\mathbf{G}: \{DMA_{en} \wedge (DMA_{addr} \in CTR) \rightarrow reset\} \tag{3.16}$$

LTL Specification (3.15) ensures that regular software does not modify $\mathcal{P}$rv's counter, while (3.16) ensures that the same is not possible via the DMA controller. FSMs in Figures 3.7 and 3.10, corresponding to `HW-Mod` access control and DMA sub-modules, must be modified to transition into *Reset* state according to these new conditions. In addition, LTL Specification (3.7) must be relaxed to allow `SW-Att` to write to CTR. Implementation and verification of the modified version of these sub-modules are publicly available at *VRASED*'s repository [1] as an optional part of the design.

3.10 Appendix: FPGA Deployment and Sample Application

VRASED can be synthesized and deployed in real IoT/CPS environments. To demonstrate its practicality and ease of use we provide, as part of *VRASED*'s repository [1], a ready-to-go synthesize-able version of the architecture for the commodity FPGA Basys3 [8] (depicted in Figure 3.14). The design can be easily ported to other FPGA models by mapping the input and output ports accordingly in the Verilog constraints file.

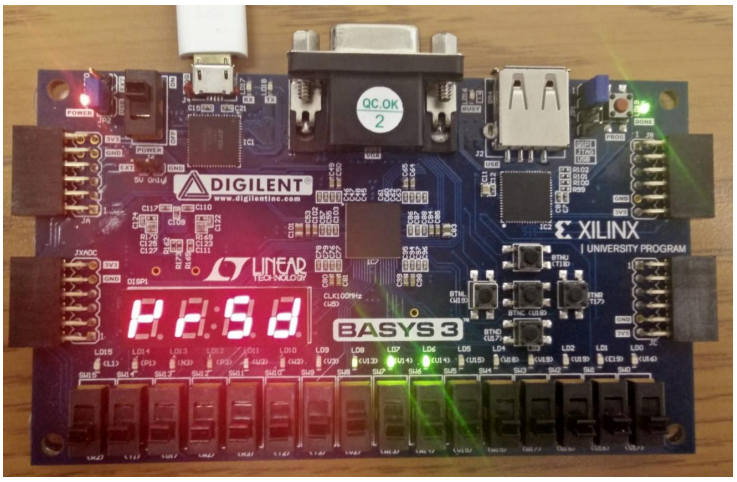


Figure 3.14: Basys3 FPGA running *VRASED*'s HW architecture depicted in Figure 3.2

Figure 3.15 presents a toy sample application written in MSP430 C. In it, $P3$ is an 8-bit General Purpose Input Output (GPIO) port which, in the synthesized HW, is connected to LEDs 0-7 of Basys3 FPGA. Lines 2-4 allocate buffers for the attestation challenge and response and initialize the challenge buffer. In practice, the challenge is received from $\mathcal{V}$rf via communication channels such as MSP430 Universal Asynchronous Receiver/Transmitter (UART). For the sake of clarity and brevity we omit the communication step from the example and set the challenge to a constant. Line 6 in Figure 3.15 sets P3 GPIO as output (i.e., an actuator port) and line 7 initializes all 8 bits to zero, making all LEDs initially off.

[8] https://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/

```
1  int main() {
2    uint8_t challenge[32];
3    uint8_t response[32];
4    my_memset(challenge, 32, 1);
5
6    P3DIR  =  0xFF;
7    P3OUT  =  0x00;
8    uint32_t count = 0;
9    volatile uint8_t buffer = 0;
10   while (1) {
11      while (count < 3000000) {
12         count ++;
13      }
14      count = 0;
15      P3OUT++;
16      if (P3OUT % 10 == 0) {
17         buffer = P3OUT;
18         P3OUT = 0xFF;
19         VRASED(challenge, response);
20         count = 0;
21         P3OUT = buffer;
22     }
23
24   }
25   return 0;
26 }
```

Figure 3.15: Toy MSP430 application demo running *VRASED*'s RA in real HW

The main application loop starts at line 10; at every iteration an artificial delay of 3 million integer increments is introduced and then P3 output value is incremented. This results in a binary counter being displayed on the 8 LEDs. At every time the counter value reached a multiple of 10 (line 16), all LEDs turn on (line 18) and the RA procedure is called in line 19 (by default, VRASED computes an attestation result over the entire program memory, although the range is configurable and allows for data memory attestation as well). The LEDs remain on until the end of RA computation. After completion of attestation, the attestation result is saved in the `response` buffer and the counter resumes. In practical applications, the result can be reported back to $\mathcal{V}$rf (e.g., via UART). A demo video of this application running on real hardware and computing RA in a small fraction of a second is available on *VRASED*'s repository [1].

Chapter 4

RATA: Remote Attestation with TOCTOU Avoidance

Abstract

Prior RA techniques (including *VRASED*, discussed in Chapter 3) verify the remote device's binary at the time when RA functionality is executed, thus providing no information about the device's binary before current RA execution or between consecutive RA executions. This implies that presence of transient malware (in the form of modified binary) may be undetected. In other words, if transient malware infects a device (by modifying its binary), performs its nefarious tasks, and erases itself before the next attestation, its temporary presence **will not be detected**. This important problem, called Time-Of-Check-Time-Of-Use (TOCTOU), is well-known in the research literature and thus far remained unaddressed in the context of hybrid RA.

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

The contributions described in this chapter will appear in the Proceedings of ACM CCS – 2021 (see preprint at [90]).

4.1 Introduction

Current hybrid RA architectures share a common limitation: they measure the state of $\mathcal{P}$rv's executables at the time when RA is executed by $\mathcal{P}$rv. They provide no information about $\mathcal{P}$rv's executables **before** RA measurement or its state in **between** two consecutive RA measurements. We refer to this problem as *Time-Of-Check Time-Of-Use* or TOCTOU. While variants of this problem have been discussed before [31, 28, 52, 23, 104], it remains unsolved in the context of hybrid RA.

We emphasize that the RA–TOCTOU problem (as formulated in this chapter) is distinct from ensuring temporal consistency between attestation and execution of a binary. The latter is tackled by runtime attestation techniques, e.g., [42, 6, 43, 44] (see Chapter 5). Nonetheless, static RA (i.e., RA of binaries) is still relevant in that context because most runtime attestation techniques for low-end devices rely on it as a building block (one exception is [55], which, instead, assumes that binaries never change [1]). Additionally, as we discuss in Section 4.7, an RA architecture secure against TOCTOU makes runtime attestation techniques that rely on static RA substantially more efficient.

In practice, the TOCTOU problem leaves devices vulnerable to transient malware which erases itself after completing its tasks. This is harmful in settings where numerous MCUs report measurements collected over extended periods. For example, consider several MCU-based sensors that measure energy consumption in a smart city, where large scale erroneous measurement may lead to power outages. If regular RA schemes that are not secure against TOCTOU are used in this case (e.g., by performing RA once a day, or once per billing cycle), security can be subverted by **(i)** changing the sensor software to spoof measurements during regular usage, and **(ii)** reprogramming the sensor back with the expected executable immediately before the scheduled RA computation. In particular, since the RA request

[1] In many settings involving low-end MCUs this assumption is unrealistic. See Section 4.2.1 for details.

must be received through an *untrusted* communication channel, malware may simply erase itself upon detecting an incoming attestation request, even if RA schedule is not known *a priori*. We note that, in settings where detection of runtime violations (e.g., code-reuse and data corruption attacks) is also desirable (e.g., MCU code is written with memory-unsafe languages), TOCTOU-Security of the underlying static RA makes the overall runtime attestation more efficient (see Section 4.7).

Our approach is rooted in the observation that current hybrid RA techniques use trusted hardware only to detect security violations that compromise execution of RA software itself and take action (e.g., by resetting the device) if such a violation is detected. Whereas, $RATA$'s main new feature is the use of a minimal (formally verified) hardware component to additionally provide historical context about the state of $\mathcal{P}$rv's program memory. This is achieved via secure logging of the latest timing of program memory modifications in a protected memory region that is covered by RA's integrity-ensuring function. This enables $\mathcal{V}$rf to later check authenticity and integrity of $\mathcal{P}$rv's memory modifications. This new feature is integrated seamlessly into the underlying RA architecture and the composition is shown to be secure. We believe this results in the following contributions:

– **RA TOCTOU-Security Formulation:** We motivate and formalize TOCTOU in the context of RA. We define RA TOCTOU-Security using a security game (see Definition 8) and discuss why current RA techniques based on consecutive self-measurements do not satisfy this definition. We believe our work to be the first formal treatment of this matter. Furthermore, we evaluate practicality of RA techniques based on consecutive self-measurements and argue that using them to obtain TOCTOU-Security incurs extremely high runtime overhead, possibly starving benign applications on $\mathcal{P}$rv.

– **RATA Design, Implementation & Verification:** We propose two techniques: $RATA_A$ and $RATA_B$. The former assumes that $\mathcal{P}$rv has a secure read-only Real-Time Clock (RTC) synchronized with $\mathcal{V}$rf. Since this assumption is unrealistic for many low-end $\mathcal{P}$rv-s, we

construct $RATA_B$ which trades off the need for a secure clock for the need to authenticate $\mathcal{V}$rf's attestation requests; this feature is already included in several hybrid RA architectures.

We show that both techniques satisfy the formal definition of TOCTOU-Security, assuming that their implementations adhere to a set of formal specifications, stated in Linear Temporal Logic (LTL). Finally, the implementation itself is formally verified to adhere to these LTL specifications, yielding security at both design and implementation levels. Our implementation is publicly available at [4]. It is realized on a real-world low-end MCU – TI MSP430 – and is deployed using commodity FPGAs. Experimental results show low hardware overhead, affordable even for cost-sensitive low-end devices.

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

4.2 Problem Scope & Definitions

4.2.1 Detection, Prevention & Memory Immutability

As a detection-oriented security service, RA does not prevent future binary modifications. Therefore, the term TOCTOU should be considered in retrospective. In particular, techniques presented in this chapter allow $\mathcal{V}$rf to understand "since when" $\mathcal{P}$rv's memory remained the same as reported in the present RA result.

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

A less drastic approach is to prevent program memory modifications that occur at run-time. This approach is vulnerable to modifications by adversaries with physical access to re-program $\mathcal{P}\mathsf{rv}$ directly. More importantly, (even if attacks that require physical access are out of scope) it makes remote updates impossible, requiring physical access whenever a device's binary needs to be updated. Since these devices are often remote or physically inaccessible (inside a larger systems, e.g., a vehicle) low-end MCUs (including aforementioned MSP430 and ATMega) typically do not prevent modifications to program memory. Our detection-based approach conforms with that necessity, allowing changes to binaries and reporting them to $\mathcal{V}\mathsf{rf}$: even if they happen in between subsequent attestations. In turn, $\mathcal{V}\mathsf{rf}$ is informed about binary changes, and can distinguish illegal modifications from expected ones.

4.2.2 Device Model & MCU Assumptions

Below, we overview MCU assumptions relevant to *RATA*. They reflect the behavior of the class of low-end embedded systems discussed in Section 4.2.1 and are in accordance with previous work on securing low-end MCUs [49, 88, 40, 42, 41]. In particular, we assume that the MCU hardware correctly implements its specifications, as follows:

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

A2 – ***Memory Address:*** Whenever memory is read or written, a data-address signal

(D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, while, for a write access, a data write-enable bit (W_{en}) must be set.

A3 – *DMA:* Whenever the Direct Memory Access (DMA) controller attempts to access the main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA cannot access memory when DMA_{en} is off (logical zero).

A4 – *MCU Reset:* At the end of a successful reset routine, all registers (including PC) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine cannot be modified. When a reset happens, the corresponding *reset* signal is set. The same signal is also set when the MCU initializes for the first time.

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

4.2.3 RA Definitions, Architectures & Adversary Model

Recall that RA is typically realized as a challenge-response protocol between $\mathcal{V}$rf (challenger) and $\mathcal{P}$rv, a potentially compromised remote low-end device. This notion is captured by a

generic syntax for RA protocols in Definition 6[2].

Definition 6 (syntax)**.** RA *is a tuple (***Request***,* **Attest***,* **Verify***) of algorithms:*

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

Note: *In the parameter list,* $(\cdots)$ *denotes that additional parameters might be included, depending on the specific* RA *construction.*

Definition 6 specifies RA as a tuple (**Request**, **Attest**,**Verify**). **Request** is computed by $\mathcal{V}\mathsf{rf}$ to produce challenge $\mathcal{C}\mathsf{hal}$ and send it to $\mathcal{P}\mathsf{rv}$. **Attest** is performed by $\mathcal{P}\mathsf{rv}$ by using $\mathcal{C}\mathsf{hal}$ to compute an authenticated integrity-ensuring function (e.g., MAC) over attested memory range (denoted by AR) and producing H, which is sent back to $\mathcal{V}\mathsf{rf}$ for verification. For example, if **Attest** is implemented using a MAC, H is computed as:

$$\mathsf{H} = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), AR) \tag{4.1}$$

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

Although techniques discussed in this chapter are not tied to a specific RA architecture, we chose to compose $RATA$ with *VRASED* due to *VRASED*'s formal security definitions, which allow reasoning about $RATA$'s secure composition with the underlying RA architecture.

[2]For simplicity, this definition omits a typical provisioning/initialization phase, in which $\mathcal{P}\mathsf{rv}$ is assigned a key and has the RA code imprinted/burned into ROM.

For instance, Theorems 3 and 4 are proven by reduction using *VRASED*'s security game (reviewed in Definition 7).

Definition 7. *VRASED's Security Game (Adapted from [40])*
Notation:
- l is the security parameter and $|\mathcal{K}| = |\mathsf{Chal}| = |MR| = l$
- $AR(t)$ denotes the content of AR at time t
RA-*game:*

1. **Setup***: $\mathcal{A}\mathsf{dv}$ is given oracle access to* **Attest** *(SW-Att) calls.*
2. **Challenge***: A challenge $\mathcal{C}\mathsf{hal}$ is generated by calling* **Request** *(Definition 6) and given to $\mathcal{A}\mathsf{dv}$.*
3. **Response***: $\mathcal{A}\mathsf{dv}$ responds with a pair (M, σ), where σ is either forged by $\mathcal{A}\mathsf{dv}$, or is the result of calling* **Attest** *(Definition 6), at some arbitrary time t.*
4. *$\mathcal{A}\mathsf{dv}$ wins iff $M \neq AR(t)$ and $\sigma = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), M)$.*

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

Recall that *VRASED* guarantees that no probabilistic polynomial time (PPT) adversary wins the RA *security game* in Definition 7 with non-negligible probability in the security parameter l, i.e., $Pr[\mathcal{A}\mathsf{dv}, \text{RA-game}] \leq \mathsf{negl}(l)$.

Remark 1: *While aforementioned guarantees ensure consistency of attested memory during attestation computation, VRASED or any prior low-end* RA *scheme is not TOCTOU-Secure, as modifications before attestation remain undetected.*

Adversary Model. The adversary model considered in this chapter remains the same as Chapter 3. We review it below.

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

Even though $\mathcal{A}\mathsf{dv}$ may physically re-program $\mathcal{P}\mathsf{rv}$'s software through wired connection to

flash, invasive/tampering hardware attacks are out of scope: we assume that $\mathcal{A}\mathsf{dv}$ can not: (1) alter hardware components, (2) modify code in ROM, (3) induce hardware faults, or (4) retrieve $\mathcal{P}\mathsf{rv}$ secrets via physical side-channels. Protection against physical hardware attacks is orthogonal to our goals and attainable via tamper-resistance techniques [95].

4.3 RA TOCTOU

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

4.3.1 Notation

We summarize our notation in Table 4.1. We keep it mostly consistent with Chapter 3, with a few additional elements to denote $RATA$-specific memory regions and signals. To simplify the notation, when the value of a given signal (e.g., D_{addr}) is within a certain range (e.g., $AR = [AR_{min}, AR_{max}]$), we write that $D_{addr} \in AR$, i.e.:

$$D_{addr} \in AR \quad \equiv \quad AR_{min} \leq D_{addr} \leq AR_{max} \tag{4.2}$$

In conformance with axioms discussed in Section 4.2.2, we use $Mod\_Mem(x)$ to denote a modification to memory address address x. Given our machine model, the following logical equivalence holds:

$$Mod\_Mem(x) \equiv (W_{en} \wedge D_{addr} = x) \vee (DMA_{en} \wedge DMA_{addr} = x) \tag{4.3}$$

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

$$Mod\_Mem(R) \equiv (W_{en} \wedge D_{addr} \in R) \vee (DMA_{en} \wedge DMA_{addr} \in R) \tag{4.4}$$

Table 4.1: Summary of $RATA$-relevant notation

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

4.3.2 TOCTOU-Security Definition

Definition 8 captures the notion of TOCTOU-Security. In it, the game formalizes the threat model discussed in Section 4.2.3, where $\mathcal{A}$dv controls $\mathcal{P}$rv's entire software state, including the ability to invoke **Attest** at will. The game starts with the challenger ($\mathcal{V}$rf) choosing a time t_0. At a later time (t_{att}), $\mathcal{A}$dv receives $\mathcal{C}$hal and wins the game if it can produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$

that is accepted by **Verify** as a valid response for expected AR value M, when, in fact, there was a time between t_0 and t_{att} when $AR \neq M$.

Definition 8.
8.1 RA-***TOCTOU Security Game:*** *Challenger plays the following game with* $\mathcal{A}\mathsf{dv}$*:*

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

$$\mathbf{Verify}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{C}\mathsf{hal}, M, \cdots) = 1 \tag{4.5}$$

and

$$\exists_{t_0 \leq t_i \leq t_{att}} \{AR(t_i) \neq M\} \tag{4.6}$$

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

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

$$Pr[\mathcal{A}\mathsf{dv}, \text{RA-}TOCTOU\text{-}game] \leq \mathsf{negl}(l)$$

where l *is the security parameter.*

This definition augments RA security (Definition 7) to incorporate TOCTOU attacks, by additionally allowing $\mathcal{A}\mathsf{dv}$ to win if it can produce the expected response and AR was modified at any point after t_0, where t_0 is chosen by $\mathcal{V}\mathsf{rf}$. For example, if $\mathcal{V}\mathsf{rf}$ wants to know if AR remained in a valid state for the past two hours, $\mathcal{V}\mathsf{rf}$ chooses t_0 as $t_0 = t_{att} - 2h$. Note that this definition also captures security against transient attacks wherein $\mathcal{A}\mathsf{dv}$ changes modified memory back to its expected state and leaves the device, thus attempting to hide its ephemeral modification from the upcoming attestation request. This attack is undetectable by all RA schemes that are not TOCTOU-Secure.

Remark 2: *Recall that, in the context of this chapter, AR corresponds to the executable part of* $\mathcal{P}\mathsf{rv}$*'s memory, i.e., program memory. Since data memory is not executable (see Section 4.2.2), changes to data memory are not taken into account by Definition 8. RATA's relation to runtime/data-memory attacks is discussed in Section 4.7.4.*

4.3.3 TOCTOU-Secure RA vs. Consecutive Self-Measurements

RA schemes based on consecutive self-measurements [31, 61] attempt to detect transient malware that comes and goes between two successive RA measurements. The strategy is for $\mathcal{P}$rv to intermittently (based on an either periodic or unpredictable schedule) and unilaterally invoke its RA functionality. Then, either $\mathcal{P}$rv self-reports to $\mathcal{V}$rf [61], or it accumulates measurements locally and waits for $\mathcal{V}$rf to explicitly request them [31]. Upon receiving RA response(s), $\mathcal{V}$rf checks for malware presence at the time of each RA measurement. Time intervals used in these RA schemes are depicted in Figure 4.1.

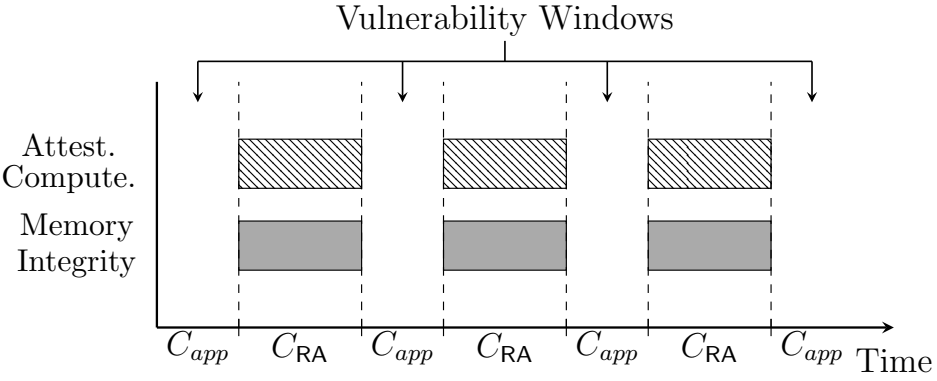


Figure 4.1: Consecutive Self-Measurements

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

$$U = \frac{C_{app}}{C_{app} + C_{\mathsf{RA}}} \tag{4.7}$$

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

$$C_{app} < C_{\mathcal{A}\mathsf{dv}} \tag{4.8}$$

where $C_{\mathcal{A}\mathsf{dv}}$ is the hypothetical number of instruction cycles used by the fastest transient malware, capable of infecting $\mathcal{P}\mathsf{rv}$, performing its tasks, and erasing itself. To illustrate this point, we assume a conservative number for $C_{\mathcal{A}\mathsf{dv}}$ to be 10^6 cycles. In this case:

$$C_{\mathcal{A}\mathsf{dv}} = 10^6 \implies C_{app} < 10^6 \implies U < \frac{10^6}{10^6 + C_{\mathsf{RA}}} \tag{4.9}$$

For example with C_{RA}, consider the number of CPU cycles required by *VRASED* (other hybrid RA architectures, e.g., [49], have similar costs) to attest a program memory of 4KB: $C_{\mathsf{RA}} = 3.6 \times 10^6$ CPU cycles (about half a second in a typical 8MHz low-end MCU).

$$U < \frac{10^6}{10^6 + 3.6 \times 10^6} \implies U < 21.74\% \tag{4.10}$$

To detect transient malware, a large fraction of CPU cycles (almost 80% in this toy example) is spent on RA computation. In practice, it is hard to determine $C_{\mathcal{A}\mathsf{dv}}$ and, in some cases (e.g., changing a general-purpose input/output value to trigger actuation), it is likely to be much less than 10^6 cycles, resulting in even lower CPU utilization left for legitimate applications running on $\mathcal{P}\mathsf{rv}$. Therefore, detection of all transient malware using consecutive self-measurements is impractical. This also applies to the case where the interval between successive measurements is variable and/or randomly selected from a range $[0, t_{max}]$. As discussed in [61], this is because it must be that $t_{max} < C_{\mathcal{A}\mathsf{dv}}$ in order to achieve negligible probability of malware evasion.

As shown in Figure 4.2, TOCTOU-Secure RA (per Definition 8) allows $\mathcal{V}\mathsf{rf}$ to ascertain memory integrity independently from the time between successive RA measurements, regardless of

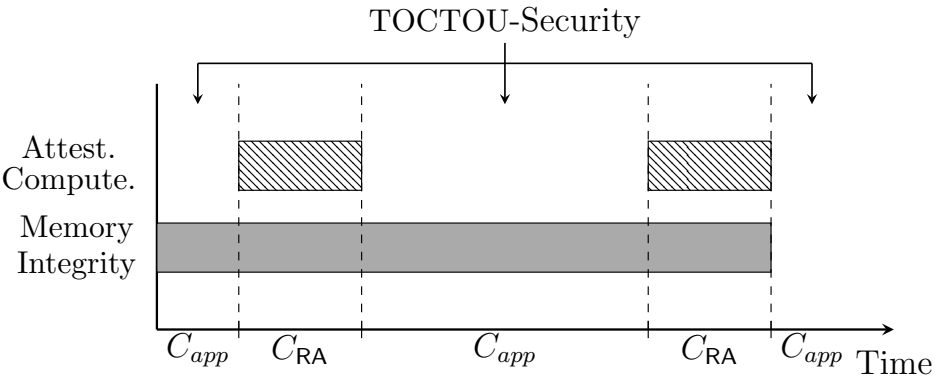


Figure 4.2: TOCTOU-Secure RA

transient malware's speed. In the next sections, we propose two TOCTOU-Secure techniques and show their security with respect to Definition 8.

4.4 $RATA_A$: RTC-Based TOCTOU-Secure Technique

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

We now overview $RATA_A$– a simple technique that requires $\mathcal{P}$rv to have a reliable read-only Real-Time Clock (RTC) synchronized with $\mathcal{V}$rf. However, RTCs are not readily available on low-end MCUs and secure clock synchronization in distributed systems is challenging [13, 11, 86], especially for low-end embedded systems [46, 53]. Nonetheless, we start with this simple approach to show the main idea behind TOCTOU-Secure RA. Next, Section 4.5 proposes an alternative variant that removes the RTC requirement, as long as $\mathcal{V}$rf requests are authenticated by $\mathcal{P}$rv. Note that $\mathcal{V}$rf authentication is already included in some current

hybrid RA architectures, including *VRASED*.

4.4.1 $RATA_A$: Design & Security

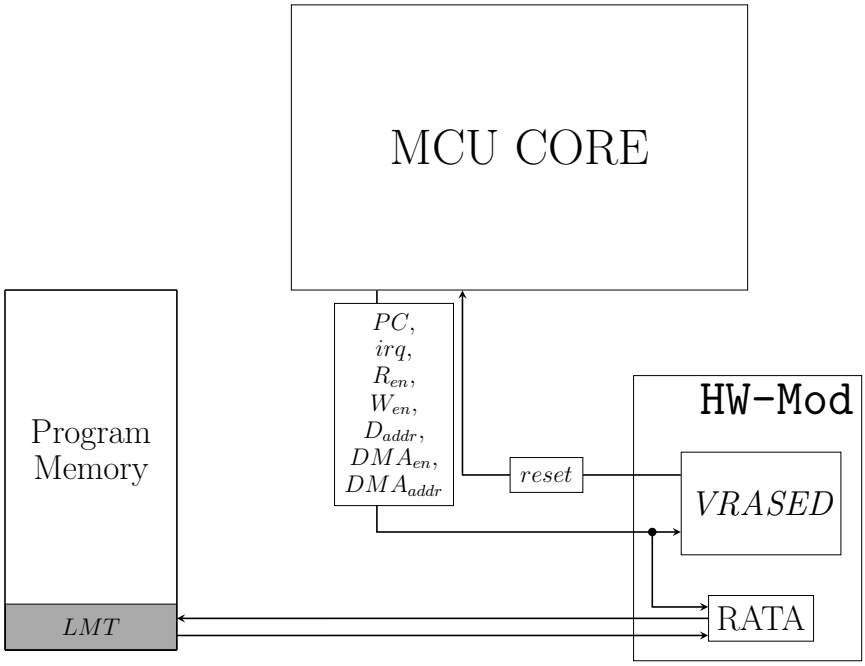


Figure 4.3: $RATA$ module in the overall system architecture

$RATA_A$ is illustrated in Figure 4.3; it is designed as a verified hardware module behaving as follows:

(1) It monitors a set of CPU signals and detects whenever any location within AR is written. This is achieved by checking the value of signals D_{addr}, W_{en}, DMA_{addr}, and DMA_{en} (see Section 4.2.3). These signals allow for detection of memory modifications either by CPU or by DMA.

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

(3) In the memory layout, $LMT \in AR$. Also, $RATA_A$ enforces that LMT is always

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

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

$$\textbf{\textit{Formal statement (LTL):}} \quad \mathbf{G}\{Mod\_Mem(LMT) \rightarrow reset\} \tag{4.11}$$

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

$$\textbf{\textit{Formal statement (LTL):}} \quad \mathbf{G}\{Mod\_Mem(AR) \leftrightarrow set_{LMT}\} \tag{4.12}$$

  where $reset$ is a 1-bit signal that triggers an immediate reset of the MCU, and set_{LMT} is a 1-bit output signal of $RATA_A$ controlling the value of LMT reserved memory. Whenever $set_{LMT} = 1$, LMT is updated with the current value from the real-time clock (RTC). LMT maintains its previous value otherwise.
- **Verify**$^{\mathcal{V}\mathsf{rf}}(\mathsf{H}, \mathcal{C}\mathsf{hal}, M, t_0, t_{LMT})$*: t_0 is an arbitrary time chosen by $\mathcal{V}\mathsf{rf}$, as in Definition 8. Upon receiving $t_{LMT}||\mathsf{H}$ $\mathcal{V}\mathsf{rf}$ checks:*

$$t_{LMT} < t_0 \tag{4.13}$$

$$\mathsf{H} \equiv HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), M) \tag{4.14}$$

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

read-only for all software executing on the MCU, and for DMA.

Note that, by enforcing $LMT \in AR$, the attestation result $\mathsf{H} = HMAC(KDF(\mathcal{K}, MR), AR)$ includes the authenticated value of LMT – the time corresponding to the latest modification of AR. As part of the **Verify** algorithm, $\mathcal{V}\mathsf{rf}$ compares this information with the time of the last authorized modification (t_0 of Definition 8) of AR to check whether any unauthorized modifications occurred since then. The general idea is further specified in Construction 1, which shows how $RATA_A$ can be seamlessly integrated into *VRASED*, enforcing two additional properties in hardware to obtain TOCTOU-Security. These properties are formalized in LTL in Equations 4.11 and 4.12 of Construction 1.

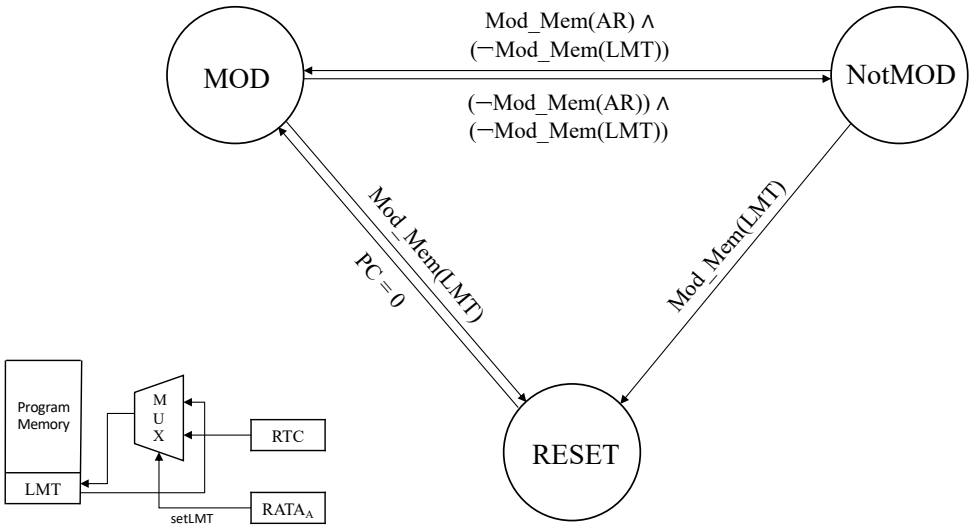


Figure 4.4: $RATA_A$ FSM for RTC-based TOCTOU-secure RA

We show that Construction 1 is secure as long as $RATA_A$ implementation adheres to LTL statements in Equations 4.11 and 4.12. This verification is discussed in Section 4.4.2. The cryptographic proof is by reduction from *VRASED* security (per Definition 7) to TOCTOU-Security (per Definition 8) of Construction 1. For its part, *VRASED* is shown secure according to Definition 7 as long as HMAC is a secure, i.e., existentially unforgeable [78], MAC (as discussed in Chapter 3). The proof of Theorem 3 is presented in Appendix 4.10.

Theorem 3. *Construction 1 is TOCTOU-Secure according to Definition 8 as long as VRASED is secure according to Definition 7.*

4.4.2 $RATA_A$: Implementation & Verification

Construction 1 (and respective security proof) assumes that properties in Equations 4.11 and 4.12 are enforced by $RATA_A$. Figure 4.4 shows a formally verified FSM corresponding

to this implementation. It enforces two properties of Equations 4.11 and 4.12. This FSM is implemented as a Mealy machine, where output changes anytime based on both the current state and current input values. The FSM takes as an input a subset of signals, shown in Figure 4.3, and produces two 1-bit outputs: *reset* to trigger an immediate reset and set_{LMT} to control the value of LMT memory location (see Construction 1). *reset* is 1 whenever FSM transitions to $RESET$ state and while it remains in that state; it remains 0 otherwise. Whereas, set_{LMT} is 1 when FSM transitions to MOD state, and becomes 0 whenever it transitions out of MOD state. $set_{LMT} = 0$ in all other cases.

The FSM works by monitoring write access to LMT and transitioning to $RESET$ whenever such attempt happens. When the system is running (i.e., $reset = 0$), FSM also monitors write access to AR and transitions to MOD state whenever it happens. The FSM transitions back to $NotMOD$ state if AR is not being modified. We design the FSM in Verilog HDL and automatically translate into SMV using Verilog2SMV [65]. Finally, we use NuSMV model checker [35] to prove that the FSM complies with invariants 4.11 and 4.12. The implementation and correspondent verification are available in [4].

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

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

4.5 $RATA_B$: Clockless TOCTOU-Secure RA Technique

We now describe $RATA_B$: a TOCTOU-Secure technique that requires no clock on $\mathcal{P}$rv. We apply the ideas from $RATA_A$ by using hardware to convey authenticated information about *the time of the latest memory modification* as part of the attestation result. However, lack of RTC precludes any notion of "time" on $\mathcal{P}$rv's end. To cope with this, we rely on $\mathcal{V}$rf to convey information tied to a given point in time, according to $\mathcal{V}$rf's own local clock. This is done as a part of RA **Request** algorithm. In fact, $RATA_B$ uses the attestation challenge ($\mathcal{C}$hal) itself in this task, taking advantage of the fact that $\mathcal{C}$hal is *unique* per **Request** and is available in any RA technique, thus incurring no additional communication overhead. Security of $RATA_B$ is tightly coupled with authentication of $\mathcal{V}$rf **Request**, which is supported by *VRASED* architecture [40].

4.5.1 $RATA_B$ – Design & Security

The design of $RATA_B$ remains consistent with Figure 4.3. $RATA_B$ monitors the same set of MCU signals as $RATA_A$ and also works by overwriting the special memory region $LMT \in AR$. However, instead of logging an RTC timestamp to LMT, it logs $\mathcal{C}$hal, which was sent by $\mathcal{V}$rf as a part of its **Request** and given as input to **Attest**($\mathcal{C}$hal, ...). LMT is overwritten with the currently received $\mathcal{C}$hal if and only if, a modification of AR occurred since the previous **Attest** instance. In summary, $RATA_B$ security relies on the following properties, enforced by its verified hardware implementation (see Section 4.5.2):

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

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

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

- **Request**$^{\mathcal{V}\mathsf{rf} \rightarrow \mathcal{P}\mathsf{rv}}()$*: $\mathcal{V}\mathsf{rf}$ generates a pair $[\mathsf{Chal}, \mathcal{A}\mathsf{uth}]$ according to VRASED authentication algorithm (see Appendix 3.9 for details) and sends it $\mathcal{P}\mathsf{rv}$.*
- **Attest**$^{\mathcal{P}\mathsf{rv} \rightarrow \mathcal{V}\mathsf{rf}}(\mathsf{Chal}, \mathcal{A}\mathsf{uth})$*: Upon receiving $[\mathsf{Chal}, \mathcal{A}\mathsf{uth}]$, $\mathcal{P}\mathsf{rv}$ behaves as follows:*
  1. *Call VRASED RA function to use $\mathcal{A}\mathsf{uth}$ to authenticate Chal. If authentication succeeds, proceed to next step. Otherwise, ignore the request.*
  2. *Compute $\mathsf{H} = HMAC(KDF(\mathcal{K}, \mathsf{Chal}), AR)$.*
  3. *Send $LMT||\mathsf{H}$ to $\mathcal{V}\mathsf{rf}$.*

  To support this operation, at all times, $RATA_B$ hardware on $\mathcal{P}\mathsf{rv}$ enforces the following:
  – LMT is read-only to software:

$$\textbf{\textit{Formal statement (LTL):}} \quad \mathbf{G}\{Mod\_Mem(LMT) \rightarrow reset\} \tag{4.15}$$

  – LMT is never updated without authentication:

$$\textbf{\textit{Formal statement (LTL):}} \quad \mathbf{G}\{[\neg UP_{LMT} \wedge \mathbf{X}(UP_{LMT})] \rightarrow \mathbf{X}(PC = CR_{auth})\} \tag{4.16}$$

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

$$\textbf{\textit{Formal statement (LTL):}}$$
$$\mathbf{G}\{Mod\_Mem(AR) \vee reset \rightarrow [(PC = CR_{auth} \rightarrow UP_{LMT})\ \mathbf{W}\ (PC = CR_{max} \vee reset)]\} \tag{4.17}$$

  where $reset$ is a 1-bit signal that triggers an immediate reset of the MCU, and UP_{LMT} is a 1-bit signal that, when set to 1, replaces the content of LMT with the current value stored in MR region (i.e., Chal). LMT maintains its previous value otherwise.
- **Verify**$^{\mathcal{V}\mathsf{rf}}(\mathsf{H}, \mathsf{Chal}, M, t_0, P, LMT)$*: Let t_0 denote a time chosen by $\mathcal{V}\mathsf{rf}$, as in Definition 8. Denote the current values in the challenge-time association pair stored by $\mathcal{V}\mathsf{rf}$ as $P = (\mathsf{Chal}_P, t_P)$. Upon receiving $LMT||\mathsf{H}$, $\mathcal{V}\mathsf{rf}$ behaves as follows:*
  1. *Check if $\mathsf{H} \equiv HMAC(KDF(\mathcal{K}, \mathsf{Chal}), M)$, where M is the expected AR value. Since AR includes LMT, M is set to contain the value of LMT, as received from $\mathcal{P}\mathsf{rv}$. Hence, this checks also assures integrity of LMT in AR. If this check fails,* **return 0***, otherwise, proceed to step 2;*
  2. *If $LMT = \mathsf{Chal}_P$ and $t_0 > t_P$,* **return 1***, otherwise, proceed to step 3;*
  3. *Set $P = (LMT, current\_time)$ and* **return 0***;*

(3) The first successful authentication happening after a modification of AR always causes LMT to be updated with the current value of $\mathcal{C}\mathsf{hal}$ which is stored in MR. (Recall from Table 4.1 that MR is the memory location from which **Attest** reads the value of $\mathcal{C}\mathsf{hal}$.)

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

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

– **Uniqueness of** LMT must be enforced, e.g., by having $\mathcal{V}\mathsf{rf}$ randomly sample $\mathcal{C}\mathsf{hal}$ from a sufficiently large space or use $\mathcal{C}\mathsf{hal}$ as a monotonically increasing counter, depending on specifics of **Request** algorithm. If $\mathcal{C}\mathsf{hal}$ is reused after n instances of **Request**, $\mathcal{A}\mathsf{dv}$ can wait for the n-th authentic **Request** to complete, infect $\mathcal{P}\mathsf{rv}$, perform its tasks, and leave $\mathcal{P}\mathsf{rv}$ before the $(n+1)$-st **Request** occurs (with a reused $\mathcal{C}\mathsf{hal}$), resulting in a valid response and compromised TOCTOU-Security. For example, if we use LMT as a dirty-bit (instead of $\mathcal{C}\mathsf{hal}$), security can be subverted in two **Request**-s, even if they are properly authenticated.

$RATA_B$ is specified in Construction 2. $\mathcal{P}$rv's hardware module controls the value of a 1-bit signal UP_{LMT}. When set to 1, UP_{LMT} updates LMT with the current value of MR; otherwise, LMT maintains its current value. $RATA_B$ hardware detects successful authentication of $\mathcal{V}$rf by checking whether the program counter PC points to the instruction reached immediately after successful authentication. Note that the instruction at location CR_{auth} is never reached **unless** authentication succeeds. Note that, unlike $RATA_A$, $\mathcal{V}$rf in $RATA_B$ learns whether a modification occurred since a previous successful attestation response, though not the exact time of that modification. $RATA_B$ security is stated in Theorem 4.

Theorem 4. *Construction 2 is TOCTOU-Secure according to Definition 8 as long as VRASED is secure according to Definition 7.*

Proof of Theorem 4 is deferred to Appendix 4.11.

4.5.2 $RATA_B$: Implementation & Verification

Proof of Theorem 4 assumes that $RATA_B$ hardware adheres to properties in Equations 4.15 to 4.17. Figure 4.5 shows $RATA_B$ implementation as an FSM formally verified to adhere to these properties. It takes as input a subset of signals, shown in Figure 4.3 and outputs two 1-bit signals: *reset* triggers an immediate system-wide reset and UP_{LMT} controls updates to LMT region. $UP_{LMT} = 1$ whenever the FSM transitions to state $UPDATE$ and has value 0 in all other states. $reset = 1$ whenever the FSM transitions to state $RESET$ and remains unchanged while in this state; it remains 0 otherwise. The FSM operates as follows:

1. If a software modification of LMT is attempted, FSM triggers *reset* immediately, regardless of what state it is in.
2. If no modifications are made to AR since the previous computation of **Attest**, FSM remains in $NotMOD$ state.

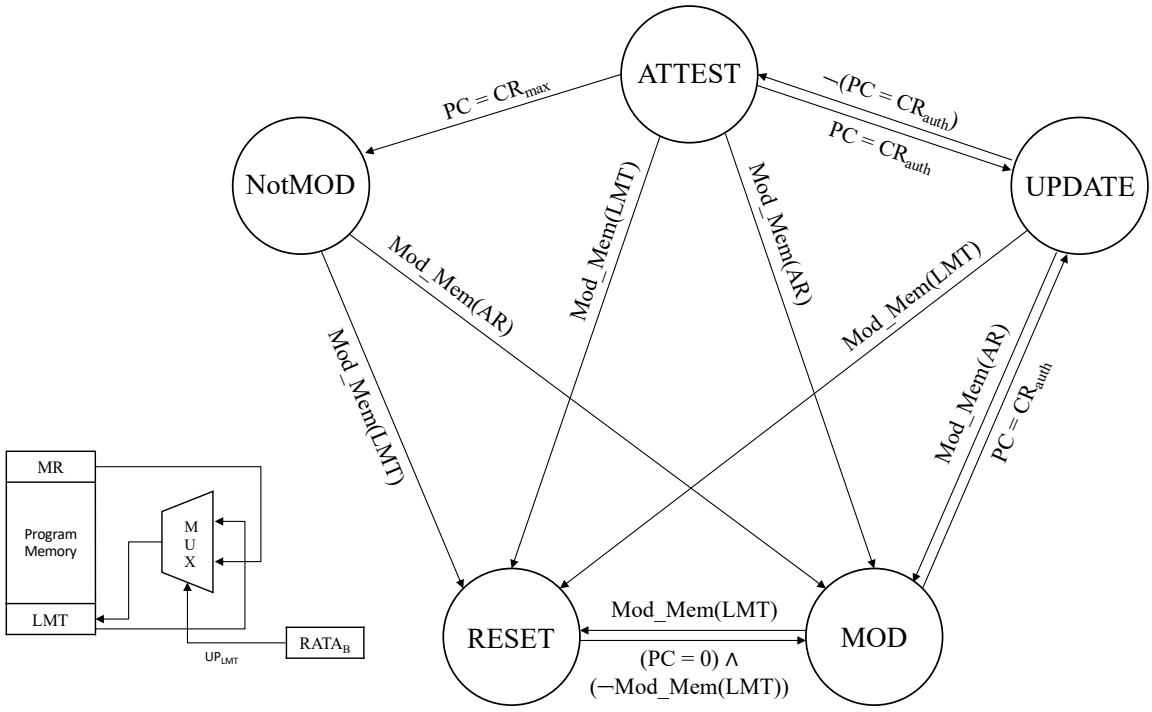


Figure 4.5: $RATA_B$ FSM for clock-less TOCTOU-secure RA

3. At any point in time, if a modification to AR is detected, FSM transitions to state MOD. This transition indicates that a modification occurred, although it neither alters any output, nor modifies LMT. This is because the information to be written to LMT (the value of $\mathcal{C}$hal in the next **Request**) is not available at this time.
4. When a call to **Attest** is made, two possible actions can occur:
   (a) If FSM is in $NotMOD$ state, **Attest** is computed normally and FSM remains in the same state.
   (b) Otherwise, FSM stays in MOD state until condition $PC = CR_{auth}$ is met, implying successful authentication of $\mathcal{V}$rf **Request**. Then, FSM transitions to state $UPDATE$ causing UP_{LMT} to be set during the transition. Hence, LMT is overwritten with $\mathcal{C}$hal passed as a parameter to the current **Attest** call. Note that update to LMT happens before the computation of the integrity-ensuring function (HMAC) over AR, which happens in state $ATTEST$. Therefore, attestation

result H will reflect $LMT = \mathcal{C}\mathsf{hal}$ as part of AR. Once **Attest** is completed ($PC = CR_{max}$), FSM transitions back to $NotMOD$.

The same verification tool-chain discussed in Section 4.4.2 is used to prove that this FSM adheres to LTL statements in Equations 4.15, 4.16, and 4.17.

4.6 Evaluation

Similar to VRASED, we implemented both RATA variants on the TI MSP430 MCU [63]. It extends *VRASED* to enable TOCTOU detection. It is synthesized and executed using Basys3 commodity FPGA prototyping board.

Hardware Overhead. Table 4.2 reflects the analysis of $RATA$ hardware overhead. Similar to some related work [40, 116, 45, 44, 42, 41], we consider the hardware overhead in terms of additional LUTs and registers. The increase in the number of LUTs can be used as an estimate of the additional chip cost and size required for combinatorial logic, while the number of extra registers offers an estimate on state registers required by sequential logic in $RATA$ FSMs. Compared to *VRASED*, the verified implementation of $RATA_A$ module takes 4 additional registers and 13 additional LUTs, while $RATA_B$ increases the number of LUTs and registers by 57 and 27, respectively. As far as the unmodified OpenMSP430 architecture, this represents the overhead of 1.4% LUTs and 1.4% registers for $RATA_A$ and 3.8% LUTs and 4.8% registers for $RATA_B$.

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

Table 4.2: Additional hardware and verification cost

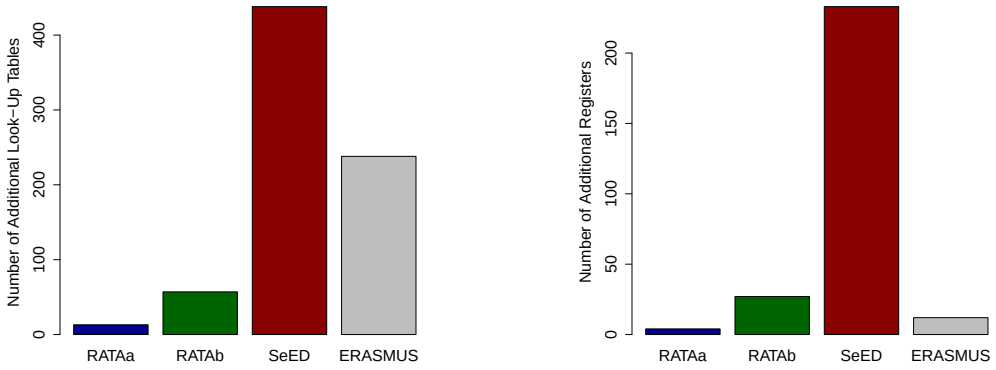
Runtime Overhead. $RATA$ does not require any modification to RA execution. It only

ensures that information about the latest modification of attested memory is factored into the attestation result. Hence, it incurs no extra runtime cycles or additional RAM allocation, on top of that of *VRASED* architecture. In fact, as we discuss next, in Section 4.7, **Attest** runtime can be reduced to the time to attest only LMT. The runtime reduction is presented in Figure 4.7. This represents a reduction of ≈ 10 times compared, e.g., to the number of cycles to attest an AR of size 4KBytes. The runtime savings increase linearly with the size of AR.

Memory Overhead. $RATA_A$ requires 128-bit of additional storage: 64 bits for RTC and 64 bits for LMT. RTC is implemented using a 64-bit memory cell incremented at every clock cycle. This guarantees that RTC does not wrap around during $\mathcal{P}$rv's lifetime since it would take more than $70,000$ years for that to happen on MSP430 running at 8MHz and incrementing RTC at every cycle. In $RATA_A$, LMT is implemented as a 64-bit memory storage and updates its content with RTC value whenever set_{LMT} bit is on. For $RATA_B$, the memory overhead increases to a total of 512 bits. 256 bits of memory are required by the implementation of *VRASED* authentication module, while another 256 bits are used to implement LMT that updates its content with $\mathcal{C}$hal when applicable (as described in Section 4.5). This small reserved memory corresponds to 0.1% of MSP430 memory address space (64KBytes in total).

Verification resources. We verify $RATA$ on an Ubuntu 18.04 machine running at 3.40GHz. Results are shown in Table 4.2. $RATA_A$ adds 127 lines of verified Verilog code on top of *VRASED*. These are needed to enforce 2 invariants in Equations 4.11 and 4.12. $RATA_B$ incurs 182 additional lines of verified Verilog code, needed to enforce the 3 invariants in Equations 4.15, 4.16, and 4.15. Besides that, $RATA$ verification requires checking existing *VRASED* invariants. Overall verification process takes less than one second and consumes at most 26MB of memory, making it suitable for a commodity desktop.

Comparison. We compare $RATA$'s hardware overhead with that of two recent self-



(a) Additional LUTs (b) Additional Registers

Figure 4.6: Hardware overhead. Comparison between $RATA$ and techniques based on self-measurements

measurement RA techniques: SeED [61] and ERASMUS [31]. Even though, as discussed in Section 4.3.3, these techniques do not achieve TOCTOU-Security (per Definition 8), we believe that they are the most closely related approaches to $RATA$. SeED extends a 32-bit Intel architecture, which is higher-end than TI MSP430. Whereas, ERASMUS was implemented on MSP430. Figure 4.6 compares $RATA$ to SeED and ERASMUS in terms of numbers of additional LUTs and registers. $RATA_A$ require fewer LUTs, compared to both SeED and ERASMUS. Whereas, $RATA_B$ necessitates more registers, compared to ERASMUS, it uses less LUTs than both self-measurements techniques. In summary, both $RATA$-s incur low overhead: $< 5\%$ increase for both LUTs and registers.

4.7 Using $RATA$ to Enhance RA & Related Services

We now discuss how $RATA$ can make RA and related services simpler and more efficient.

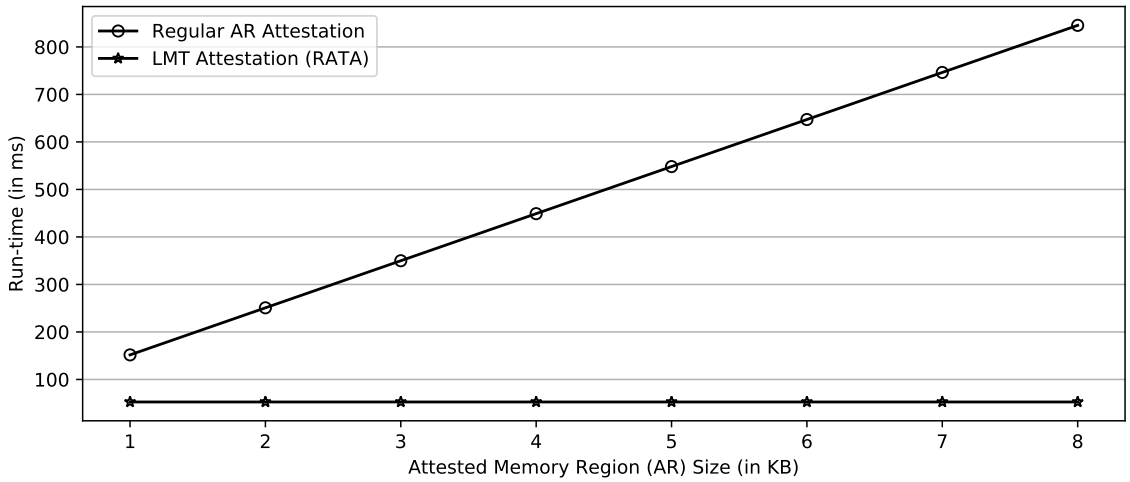


Figure 4.7: Comparison of LMT attestation time **Case-1**) with regular attestation of AR (**Case-2**), as a function of $|AR|$. $|LMT|$ is 32 Bytes. Results on the MSP430 MCU running at 8MHz.

4.7.1 Constant-Time RA

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

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

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

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

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

Most of the time, $\mathcal{P}$rv is expected to be in a benign state (i.e., no malware), especially if $\mathcal{A}$dv knows that its presence is guaranteed to be detectable. In such times, size of attested memory can be reduced reduced from several KBytes (e.g., when AR is the entire program memory on a low-end $\mathcal{P}$rv) to a mere 32 Bytes (LMT size), Figure 4.7 depicts an empirical result on the MSP430 MCU showing how this optimization can significantly reduce RA runtime overhead.

In the rest of this section, we discuss some implications of this optimization, along with security improvements offered by $RATA$, to different branches of RA and related security services.

4.7.2 Atomicity & Real-Time Settings

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

In higher-end devices, memory locking can be used to prevent modifications until the end

of attestation, as discussed in [30]. However, in low-end devices, where applications run on bare-metal and there is no architectural support for memory locking, temporal consistency is attained by enforcing that attestation software (`SW-Att`) runs atomically: once it starts, it can not be interrupted by any software running on $\mathcal{P}\mathsf{rv}$, thus preventing malware from interrupting RA and relocating itself. While effective for security purposes, this requirement conflicts with real-time requirements if $\mathcal{P}\mathsf{rv}$ serves a safety-critical and time-sensitive function.

Some prior remediation techniques proposed to enable interrupts while maintaining temporal consistency, with high probability. SMARM [32] is one such approach. (Others similar techniques are discussed in [27]). SMARM divides attested memory (AR) into a set of blocks which are attested in a randomized order. Attestation of one block remains atomic. However, interrupts are allowed between attestation of two blocks. Assuming that malware can not guess the index of the next block to be attested, even if interrupts are allowed, malware only has a certain probability of avoiding detection. If the entire attestation procedure is repeated multiple times, this probability can be made arbitrarily small.

We note that, given the $RATA$ optimization discussed in Section 4.7.1, attestation can be computed faster. In particular, since most Pseudo Random Function (PRF) implementations use block sizes of at least 32 bytes, the atomic attestation of one block in a SMARM-type strategy cannot be faster than the attestation on LMT in $RATA$ ($|LMT| = 32$ Bytes). In addition, attestation of LMT provides information about the content of AR in its entirety, with no probability of evasion. We believe this makes $RATA$ more friendly to safety-critical operations than existing approaches.

In such settings, we envision that AR would be attested in its entirety at system boot time (**Case-2** in Section 4.7.1), while subsequent RA would be computed on LMT only (**Case-1** in Section 4.7.1). We note that, if AR is eventually modified, $\mathcal{P}\mathsf{rv}$ would need to fall back to **Case-2** for the next RA computation, which takes time to run atomically. However, after an

unauthorized modification to $\mathcal{P}$rv's memory, it is unclear why one would still want to offer real-time guarantees to compromised software.

4.7.3 Collective RA Protocols and Device-to-Device Malware Relocation

Collective RA protocols (CRA) (aka swarm attestation) [16, 8, 28, 62, 69, 70, 89] are a set of techniques that attest a large number of devices that operate together as a part of a larger system. CRA schemes typically assume hybrid RA architectures on individual devices and look into how to attest many devices efficiently. One security problem that is typically out of scope on single-device RA and becomes relevant in CRA settings is caused by migratory malware. This is an analog of intra-device self-relocating malware (discussed in Section 4.7.2) that appears in collective settings. Specifically, instead of moving around inside the memory of the same device, it migrates from device to device to avoid detection.

To guarantee detection of migratory malware, CRA result must convince $\mathcal{V}$rf that all devices were in a safe state **within the same time window**, implying that malware had no destination device to which to migrate and avoid detection. Consequently, if a single-device attestation result conveys a safe state only at some point in between the execution of **Request** and **Verify** algorithms, it is nearly impossible (especially, in the presence of network delays) to conclude that migratory malware is not present in the swarm. Although this problem is discussed in the CRA literature existing approaches either place it outside their adversary model [28, 16, 8, 70], or make a strong assumption about clock synchronization among all devices in the swarm [61, 62, 69, 89], so that all devices can be scheduled to run **Attest** at the same time.

Construction 3 (CRA-$RATA$)**.** *Let $S = \{\mathcal{P}\mathsf{rv}_1, ..., \mathcal{P}\mathsf{rv}_n\}$ denote a swarm of n devices individually equipped with $RATA_B$ hybrid* RA *facilities. Let LMT_i be the value of LMT in $\mathcal{P}\mathsf{rv}_i$. Also,* **Verify**$(\mathcal{P}\mathsf{rv}_i)$ *denotes the verification algorithm of Construction 2 for $\mathcal{P}\mathsf{rv}_i$. Consider a protocol in which:*

1. $\mathcal{V}\mathsf{rf}$ *executes $RATA_B$ protocol, as defined in Construction 2 with each $\mathcal{P}\mathsf{rv}_i$ in parallel. Let $t(Req_i)$ denote the time when $\mathcal{V}\mathsf{rf}$ issued the request to $\mathcal{P}\mathsf{rv}_i$.*
2. $\mathcal{V}\mathsf{rf}$ *collects all responses and computes* **Verify**$(\mathcal{P}\mathsf{rv}_i)$ *for all $\mathcal{P}\mathsf{rv}_i \in S$. It then uses the values of LMT_i to learn "since when" $\mathcal{P}\mathsf{rv}_i$ has been in a valid state. We denote this time as $t(LMT_i)$.*

We argue that, by addressing the TOCTOU problem in the single-device setting, $RATA_B$ can be utilized to construct the first CRA protocol secure against migratory malware without relying on synchronization of the entire swarm. To see why this is the case, consider Construction 3. In this construction, TOCTOU-Security on individual devices allows $\mathcal{V}\mathsf{rf}$ to conclude that each $\mathcal{P}\mathsf{rv}$ was in a valid state within a fixed time interval. Therefore, by checking the overlap in the valid interval of all $\mathcal{P}\mathsf{rv}$-s, $\mathcal{V}\mathsf{rf}$ can learn the time window in which the entire swarm was safe as a whole, or detect migratory malware when such time window does not exist. Theorem 5 states the concrete guarantee offered by Construction 3.

Theorem 5. *In Construction 3, if for all $\mathcal{P}\mathsf{rv}_i \in S$,* **Verify**$(\mathcal{P}\mathsf{rv}_i)$ *in step 2 succeeds for some $t(LMT_i)$, then it must be the case that entire S was in a valid state in the time window defined by the interval:*

$$(\boldsymbol{max}[t(LMT_1), ..., t(LMT_n)] \ , \ \boldsymbol{min}[t(Req_1), ..., t(Req_n)]) \tag{4.18}$$

assuming equation 4.18 constitutes a valid interval.

Note: (a, b) *is a valid interval if $a < b$.*

Proof. (Sketch) It follows directly from the observations that:

– Given RA-Security, for each $\mathcal{P}\mathsf{rv}_i \in S$, a valid response can not be produced before the

time when $\mathcal{P}\mathsf{rv}_i$ receives $\mathcal{C}\mathsf{hal}$, which is strictly greater than $t(Req_i)$.

– Given TOCTOU-Security, for each $\mathcal{P}\mathsf{rv}_i \in S$ with $\mathbf{Verify}(\mathcal{P}\mathsf{rv}_i) = 1$, its memory could not have been changed between $t(LMT_i)$ and the first call to **Attest** after $t(Req_i)$. □

In addition, running RATA in a CRA setting with heterogeneous devices (with different processing power and AR sizes) helps to minimize the variability in terms of $\mathcal{P}\mathsf{rv}$-s time to respond.

4.7.4 Runtime Attestation

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

Proofs of execution (PoX) for embedded systems, which were recently explored in [42] (APEX) and are the subject of Chapter 5, are used to prove that a given operation on $\mathcal{P}\mathsf{rv}$ was performed through the execution of the expected code and to verify that outputs were indeed produced by this execution. Control Flow Attestation (CFA) introduced in [6] (C-FLAT) allows $\mathcal{V}\mathsf{rf}$ to also verify whether software that executed on $\mathcal{P}\mathsf{rv}$ took a specific (or a set of) valid control path(s) enabling detection of ROP/code-reuse type attacks to vulnerable code.

We note that regular (or static) RA is a common stepping stone in these respective functionalities. In C-FLAT, OAT [105], and Tiny-CFA [43], the executable must be instrumented with specific instructions to enable CFA and RA is used to verify that such instructions were not removed or modified. Besides, even executions with the same control-flow may differ in behavior/outputs if their instructions differ. Similarly, in APEX execution is proven to

$\mathcal{V}$rf with attestation of execution metadata. However, without attesting the corresponding executable (in program memory), this proof would have no meaning other than: "some code executed successfully".

In many applications, the same executable is expected to remain in memory for long periods of time, while its proper execution (or control-flow) must be verified repeatedly, per safety-critical embedded operation [105]. *RATA*'s optimization discussed in 4.7.1 can minimize the overhead of such successive runtime attestations.

To illustrate this concept we combined *RATA* with APEX and Tiny-CFA (which itself is implemented atop APEX). In APEX, all runtime overhead *vis-a-vis* cost of executing the same software without proving its successful execution to $\mathcal{V}$rf is caused by the cost of static RA. Since APEX is implemented atop *VRASED*, implementing an *RATA*-compliant version of APEX does not require any further modifications, aside from those described in this Chapter. (APEX details are presented in Chapter 5). As such, this approach substantially reduced PoX and CFA computational costs (these savings are consistent with Figure 4.7) while requiring the same additional hardware cost as reported in Table 4.2.

4.8 Related Work

– **Temporal Aspects of RA:** Besides TOCTOU, two other temporal aspects are essential for RA security: First, temporal consistency [30] means guaranteeing that the RA result reflects an instantaneous snapshot of $\mathcal{P}$rv's attested memory at some point in time during RA. Lack thereof allows self-relocating malware to escape detection by copying and/or erasing itself during RA. Temporal consistency is achieved by enforcing atomic (uninterruptible) execution of attestation code, or by locking attested memory (i.e., making it unmodifiable) during RA execution. Second, when RA is used on safety-critical and/or real-time devices [27], atom-

icity requirement might interfere with the real-time nature of $\mathcal{P}$rv's application. To address this issues, SMARM [32] relaxes this requirement by using probabilistic malware detection. Meanwhile, ERASMUS [31] and SeED [61] are based on $\mathcal{P}$rv's self-measurements, in order to detect transient malware that infects $\mathcal{P}$rv and leaves before the next RA instance. See Section 4.3.3 for further discussion on these types of techniques. Atrium [116] deals with physical-hardware adversaries that intercept instructions as they are fetched to the CPU during attestation. Atrium refers to that issue as TOCTOU. Despite nomenclature, that issue is clearly not the same as *RATA*'s goal.

– **Formal Verification and RA:** Formal verification provides significantly higher level of assurance, yielding provable security for protocol specifications and implementations thereof. Recently, several efforts focused on formal verification of security-critical services and systems [58, 17, 117, 19, 74, 67]. *VRASED* [40] realized a formally verified RA architecture targeting low-end devices. Other formally verified security services were obtained by extending *VRASED* to derive remote proofs of software update, memory erasure and system-wide MCU reset [41]. APEX [42] (see Chapter 5) builds on top of *VRASED* to develop a verified architecture for proofs of remote software execution on low-end devices [42]. RATA also builds on top of *VRASED*, extending it to provide TOCTOU security while retaining original verified guarantees. Relying on *VRASED* allows us to reason about *RATA* design and to formally verify its security properties. Nonetheless, *RATA*'s main concepts are applicable to other hybrid (and possibly hardware-based, such as [87]) RA architectures.

4.9 Conclusions

In this chapter, we design, prove security of, and formally verify two designs ($RATA_A$ and $RATA_B$) to secure RA against TOCTOU-related attacks, which perform binary modifications on a low-end embedded system, in between successive RA instances. $RATA_A$ and $RATA_B$

modules are formally specified and verified using a model-checker. They are also composed with *VRASED* – a verified RA architecture. We show that this composition is TOCTOU-secure using a reduction-based cryptographic proof. Our evaluation demonstrates that a TOCTOU-Secure design is affordable even for cost-sensitive low-end embedded devices. Additionally, in most cases, it reduces RA time complexity from linear to constant, in the size of the attested memory.

APPENDIX

4.10 Appendix: Proof of Theorem 3

Proof. By contradiction, assume a polynomial $\mathcal{A}\mathsf{dv}$ that wins the game in Definition 8 with probability $Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA\text{-}TOCTOU\text{-}game}] > \mathsf{negl}(l)$. Therefore, $\mathcal{A}\mathsf{dv}$ can produce $t_{LMT}||\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ such that:

$$\mathbf{Verify}^{\mathcal{V}\mathsf{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{C}\mathsf{hal}, M, t_0, t_{LMT}) = 1$$

and

$$\exists_{t_0 \leq t_i \leq t_{att}} \{AR(t_i) \neq M\}$$

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

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

$$\mathsf{H}_{\mathcal{A}\mathsf{dv}} \equiv HMAC(KDF(\mathcal{K}, MR), M) \tag{4.19}$$

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

Also, hardware enforced properties 4.11 and 4.12 guarantee that $LMT \in AR$ always contains the time of the most recent modification of AR. Thus, because $t_{att} \geq t_i$, it must be the case that $AR(t_{att})$ reflects $LMT \geq t_i$ implying $LMT \neq t_{LMT}$, and consequently $AR(t_{att}) \neq M$.

Under such restriction, $\mathcal{A}\mathsf{dv}$ ability to win the game implies its capability to produce

$\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ such that $\mathbf{Verify}^{\mathcal{V}\mathsf{rf}}(\mathsf{H}, \mathcal{C}\mathsf{hal}, M, t_0, t_{LMT}) = 1$, even though modifying AR such that $AR(t_{att}) = M$ is not possible. To conclude the proof, we show that the existence of such an $\mathcal{A}\mathsf{dv}$ implies the existence of another adversary $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ that wins the RA security game in Definition 7 against *VRASED*, contradicting the theorem's assumption.

To win the game in Definition 7 $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ behaves as follows:

1. At time t_i where $t_0 \leq t_i \leq t_{att}$, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ modifies AR causing $LMT \in AR$ to store the value of t_i.

2. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ receives $\mathcal{C}\mathsf{hal}$ from the challenger in step (2) of RA security game of Definition 7 and executes the same algorithm of $\mathcal{A}\mathsf{dv}$ with inputs $\mathcal{C}\mathsf{hal}$ and $t_{att} = t$ to produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$, such that $\mathbf{Verify}^{\mathcal{V}\mathsf{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{C}\mathsf{hal}, M, t_0, t_{LMT}) = 1$ with probability:

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

   even though $t_{LMT} < t_0 < t_i$.

3. As a response in step 3 of the game in Definition 7, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ replies with: $\sigma = \mathsf{H}_{\mathcal{A}\mathsf{dv}}$.

Since $\mathbf{Verify}^{\mathcal{V}\mathsf{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{C}\mathsf{hal}, M, t_0, t_{LMT}) = 1$, it follows that $\sigma = \mathsf{H}_{\mathcal{A}\mathsf{dv}} = HMAC(KDF(\mathcal{K}, MR), M)$, for expected M containing $LMT = t_{LMT}$. However, due to the AR modification at time t_i, $AR(t)$ must reflect $LMT \geq t_i$, satisfying the condition that $AR(t) \neq M$ and allowing $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ to win the game in Definition 7 with probability:

$$Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA}\text{-game}] = Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA}\text{-TOCTOU-game}] > \mathsf{negl}(l) \tag{4.20}$$

□

4.11 Appendix: Proof of Theorem 4

We show that, if properties in Equations 4.15, 4.16 and 4.17 hold, existence of $\mathcal{A}\mathsf{dv}$ that wins the TOCTOU security game against $RATA_B$ implies the existence of another $\mathcal{A}\mathsf{dv}$ that wins RA security game against *VRASED*, thus contradicting the initial premise.

Proof. By contradiction, assume a polynomial $\mathcal{A}\mathsf{dv}$ that wins the game in Definition 8 with probability $Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA\text{-}TOCTOU\text{-}game}] > \mathsf{negl}(l)$. Therefore, $\mathcal{A}\mathsf{dv}$ is able to produce response $LMT_{\mathcal{A}\mathsf{dv}}||\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ such that:

$$\mathbf{Verify}^{\mathcal{V}\mathsf{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{C}\mathsf{hal}, M, t_0, T, LMT_{\mathcal{A}\mathsf{dv}}) = 1$$

and

$$\exists_{t_0 \leq t_i \leq t_{att}} \{AR(t_i) \neq M\}$$

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

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

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

1. By LTL statement 4.17, any modification to AR in between the i-th and $(i+1)$-th authenticated computations of **Attest**, will cause AR to change to reflect $LMT = \mathcal{C}\mathsf{hal}_{i+1}$ in following RA responses. Therefore, the premise that

   $$\exists_{t_0 \leq t_i \leq t_{att}} \{AR(t_i) \neq M\}$$

   will necessarily update LMT.
2. From *VRASED* authentication (see Appendix 3.9), for subsequent RA challenges $\mathcal{C}\mathsf{hal}_i$ and $\mathcal{C}\mathsf{hal}_{i+1}$ that authenticate successfully, it is always the case that $\mathcal{C}\mathsf{hal}_i < \mathcal{C}\mathsf{hal}_{i+1}$.
3. From LTL statement 4.16, $RATA_B$ never updates LMT with a challenge if it does not authenticate successfully. Since authentication implies $\mathcal{C}\mathsf{hal}_i < \mathcal{C}\mathsf{hal}_{i+1}$, a call to **Attest** never causes LMT to be updated to a previously used $\mathcal{C}\mathsf{hal}$.

From observations 1, 2, and 3 above, it is impossible to set $AR = M$ by calling **Attest**, because any modification to LMT caused by **Attest** will always change LMT to a value that was never used before and thus different from $\mathcal{C}\mathsf{hal}_P$. At this point $\mathcal{A}\mathsf{dv}$ last resource is to try to write to LMT directly. However, this is immediately in conflict with LTL property 4.15.

Since making $AR(t_{att}) = M$ is impossible after a modification at time t_i, the assumption that $\mathcal{A}\mathsf{dv}$ wins the game in Definition 8 implies that $\mathcal{A}\mathsf{dv}$ is able to produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ that verifies successfully even when $AR(t_{att}) \neq M$. To conclude the proof, we show that existence of such $\mathcal{A}\mathsf{dv}$ implies existence of another adversary $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ that wins the RA security game in Definition 7.

To win the game in Definition 7 $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ is constructed as follows:

1. At time some t_i, where $t_0 \leq t_i \leq t$, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ modifies memory in AR.
2. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ receives $\mathcal{C}\mathsf{hal}$ in step 2 of RA security game of Definition 7, and executes

the same algorithm as $\mathcal{A}\mathsf{dv}$ on $\mathcal{C}\mathsf{hal}$ and with $t_{att} = t$ to produce $\mathsf{H}_{\mathcal{A}\mathsf{dv}}$ such that $\mathbf{Verify}^{\mathcal{V}\mathsf{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{C}\mathsf{hal}, M, t_0, T, LMT_{\mathcal{A}\mathsf{dv}}) = 1$ with probability:

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

3. As a response in step 3 of the game in Definition 7, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ replies with $\sigma = \mathsf{H}_{\mathcal{A}\mathsf{dv}}$.

Since $\mathbf{Verify}^{\mathcal{V}\mathsf{rf}}(\mathsf{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{C}\mathsf{hal}, M, t_0, T, LMT_{\mathcal{A}\mathsf{dv}}) = 1$, it follows that $\sigma = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), M)$ (first condition for $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ to win), for expected M containing $LMT = LMT_{\mathcal{A}\mathsf{dv}}$. On the other hand, because memory was modified at time t_i, it must be the case that $AR(t)$ has $LMT \neq LMT_{\mathcal{A}\mathsf{dv}}$. Thus satisfying the remaining condition that $AR(t) \neq M$ implying that $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ wins the game in Definition 7 with probability:

$$Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA}\text{-game}] = Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA}\text{-TOCTOU-game}] > \mathsf{negl}(l) \tag{4.21}$$

□

4.12 Appendix: *RATA* Implementation with SANCUS

To demonstrate *RATA* generality, we also implemented it atop SANCUS [87]: a hardware-based RA architecture targeting the same class of embedded devices. To the best of our knowledge, aside from VRASED (used in our verified implementation), SANCUS is the only other open-source RA architecture for low-end embedded systems, which justifies our choice. We note that this implementation is intended to demonstrate *RATA* generality and that provable security guarantees derived from *RATA*-with-*VRASED* do not apply here. Since SANCUS does not provide a formal security model and analysis, provable composition of RATA atop SANCUS is not currently possible.

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

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

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

Because SANCUS is implemented on the same MCU as *VRASED* (OpenMSP430), no internal modifications are required to $RATA$ hardware module, and its additional hardware cost remains consistent with that reported in Table 4.2. To support TOCTOU-Secure attestation of multiple SMs, this cost grows linearly, i.e., the cost incurred by one $RATA$ hardware module multiplied by the number of independent SMs that should support TOCTOU-Secure attestation. We note that, in $RATA_A$'s case, the same secure read-only synchronized clock

can be shared by all such modules.

Chapter 5

APEX: From Remote Attestation to Verified Proofs of Execution

Abstract

As discussed in previous chapters, RA offers unforgeable proofs that the expected software binary is currently installed in a remote $\mathcal{P}$rv. However, by itself, it cannot prove that this software was executed properly and it cannot prove that claimed outputs/results were indeed produced by the timely and correct execution of this software. In this chapter, we turn our attention to the problem of proving correct execution, by designing, demonstrating security of, and formally verifying, *APEX*: an Architecture for Provable Execution. To the best of our knowledge, this is the first of its kind result for low-end embedded systems. *APEX* has a range of applications, especially, authenticated sensing and trustworthy actuation, which are increasingly relevant in the context of safety-critical systems. *APEX* is publicly available and our evaluation shows that it incurs low overhead, affordable even for very low-end embedded devices. Research results described in this chapter appeared in the Proceedings of the USENIX Security Symposium – 2020 (see [42]).

5.1 Introduction

In this chapter, we show that **Proofs of Execution (PoX)** are both important and necessary for securing low-end MCUs. Specifically, PoX schemes can be used to construct sensors and actuators that "can not lie", even under the assumption of full software compromise. In a nutshell, a PoX conveys that an untrusted remote (and possibly compromised) device really executed specific software, and all execution results are authenticated and cryptographically bound to this execution. This functionality is similar to authenticated outputs that can be produced by software execution in SGX-alike architectures [64, 37] on high-end devices, such as desktops and servers.

While many RA architectures have been proposed with different assumptions and guarantees [97, 76, 49, 68, 22, 48, 50, 89, 87, 29, 40], RA **alone is insufficient to obtain proofs of execution**. RA allows $\mathcal{V}$rf to check integrity of software residing in the attested memory region on $\mathcal{P}$rv. However, by itself, RA offers no guarantee that the attested software is ever executed or that any such execution completes successfully. Even if the attested software is executed, there is no guarantee that it has not been modified (e.g., by malware residing elsewhere in memory) during the time between its execution and its attestation. This phenomenon can be viewed as a type of TOCTOU: between the RA measurement and the **execution** of the measured software (which is different from the retrospective type of TOCTOU discussed in Chapter 4). Finally, RA does not guarantee authenticity and integrity of any output produced by the execution of the attested software.

To bridge this gap, we design and implement *APEX*: an Architecture for Provable Execution. In addition to RA, *APEX* allows $\mathcal{V}$rf to request an unforgeable proof that the attested software executed successfully and (optionally) produced certain authenticated output. These guarantees hold even in case of full software compromise on $\mathcal{P}$rv. Contributions of this work include:

– **New security service:** we design and implement *APEX* for unforgeable remote proofs of execution (PoX). As discussed in the rest of this chapter, obtaining provably secure PoX requires significant architectural support on top of a secure RA functionality (see Section 5.6). Nonetheless, we show that, by careful design, *APEX* achieves all necessary properties of secure PoX with fairly low overhead. To the best of our knowledge, this is the first security architecture for PoX on low-end devices.

– **Provable security & implementation verification:** secure PoX involves considering, and reasoning about, several details which can be easily overlooked. Ensuring that all necessary PoX components are correctly implemented, composed, and integrated with the underlying RA functionality is not trivial. In particular, early RA architectures oversimplified PoX requirements, leading to the incorrect conclusion that PoX can be obtained directly from RA; see examples in Section 5.2. In this work, we show that *APEX* yields a secure PoX architecture. All security properties expected from *APEX* implementation are formally specified using Linear Temporal Logic (LTL) and *APEX* modules are verified to adhere to these properties. We also prove that the composition of *APEX* new modules with a formally verified RA architecture (*VRASED*) implies a concrete definition of PoX security.

– **Evaluation, publicly available implementation and applications:** *APEX* was implemented on a real-world low-end MCU (TI MSP430) and deployed using commodity FPGAs. Both design and verification are publicly available at [2]. Our evaluation shows low hardware overhead, affordable even for low-end MCUs. The implementation is accompanied by a sample PoX application; see Section 5.6.3. As a proof of concept, we use *APEX* to construct a trustworthy safety-critical device, whereupon malware can not spoof execution results (e.g., fake sensed values) without detection.

5.2 Related Work

RA and Execution –The first hybrid RA architecture – SMART [49] – acknowledged the importance of proving remote code execution on $\mathcal{P}$rv, in addition to just attesting $\mathcal{P}$rv's memory. Using an *attest-then-execute* approach (see Algorithm 4 in [49]), SMART attempts to provide software execution by specifying the address of the first instruction to be executed after completion of attestation. However, SMART offers no guarantees beyond "invoking the executable". It *does not guarantee that execution completes successfully* or that any produced outputs are tied to this execution. For example, SMART can not detect if execution is interrupted (e.g., by malware) and never resumed. A reset (e.g., due to software bugs, or $\mathcal{P}$rv running low on power) might happen after invoking the executable, preventing its successful completion. Also, direct memory access (DMA) can occur during execution and it can modify the code being executed, its intermediate values in data memory, or its output. SMART neither detects nor prevents DMA-based attacks, since it assumes DMA-disabled devices.

Another notable RA architecture is TrustLite [68], which builds upon SMART to allow secure interrupts. TrustLite does not enforce temporal consistency of attested memory; it is thus conceptually vulnerable to self-relocating malware and memory modification during attestation [30]. Consequently, it is challenging to deriving secure PoX from TrustLite. Several other prominent low-to-medium-end RA architectures – e.g., SANCUS [87], HYDRA [48], and TyTaN [22] – do not offer PoX. In this chapter, we show that the *execute-then-attest* approach, using a temporally consistent RA architecture, can be designed to provide unforgeable proofs of execution that are only produced if the expected software executes correctly and its results are untampered.

Control Flow Attestation (CFA)– In contrast with RA, which measures $\mathcal{P}$rv's software integrity, CFA techniques [6, 45, 116, 44] provide $\mathcal{V}$rf with a measurement of the exact control

flow path taken during execution of specific software on $\mathcal{P}$rv. Such measurements allow $\mathcal{V}$rf to detect run-time attacks. We believe that it is possible to construct a PoX scheme that relies on CFA to produce proofs of execution based on the attested control flow path. However, in this chapter, we advocate a different approach – specific for proofs of execution – for two main reasons:

- CFA requires substantial additional hardware features in order to attest, in real time, executed instructions along with memory addresses and the program counter. For example, C-FLAT [6] assumes ARM TrustZone, while LO-FAT [45] and LiteHAX[44] require a hardware branch monitor and a hash engine. We believe that such hardware components are not viable for low-end devices, since their cost (in terms of price, size, and energy consumption) is typically higher than the cost of a low-end MCU itself. For example, the hardware cost of hardware branch monitors and hash engines reported in the aforementioned CFA architectures are considerably more expensive than the low-end MCUs themselves and hence not realistic in our device context.
- CFA assumes that $\mathcal{V}$rf can enumerate a large (potentially exponential!) number of valid control flow paths for a given program, and verify a valid response for each. This burden is unnecessary for determining if a proof of execution is valid, because one does not need to know the exact execution path in order to determine if execution occurred (and terminated) successfully; see Section 5.3.1 for a discussion of run-time threats.

In contrast, we propose a PoX-specific architecture – *APEX*– that enables low-cost PoX for low-end devices. *APEX* is non-invasive (i.e., it does not modify MCU behavior and semantics) and incurs low hardware overhead: around 2% for registers and 12% for LUTs. Also, $\mathcal{V}$rf is not required to enumerate valid control flow graphs and the verification burden for PoX is nearly the same as the effort to verify a typical remote attestation response for the same code.

Formally Verified Security Services– In recent years, several efforts focused on formally

verifying security-critical systems. In terms of cryptographic primitives, Hawblitzel et al. [58] verified implementations of SHA, HMAC, and RSA. Bond et al. [21] verified an assembly implementation of SHA-256, Poly1305, AES and ECDSA. Zinzindohoué, et al. [117] developed HACL*, a verified cryptographic library containing the entire cryptographic API of NaCl [18]. Larger security-critical systems have also been successfully verified. Bhargavan [19] implemented the TLS protocol with verified cryptographic security. CompCert[74] is a C compiler that is formally verified to preserve C code semantics in generated assembly code. Klein et al. [67] designed and proved functional correctness of the seL4 microkernel. More recently, *VRASED* [40] realized a formally verified hybrid RA architecture. *APEX* architecture, proposed in this chapter, uses *VRASED* RA functionality (see Chapter 3 for details) composed with additional formally verified architectural components to obtain provably secure PoX.

Proofs of Execution (PoX)– Flicker [83] offers a means for obtaining PoX in high-end devices. It uses TPM-based attestation and sealed storage, along with late launch support offered by AMD Secure Virtual Machine extensions[112] to implement an infrastructure for isolated code execution and attestation of the executed code, associated inputs, and outputs. Sanctum [37] employs a similar approach by instrumenting Intel SGX enclaved code to convey information about its own execution to a remote party. Both of these approaches are only suitable for high-end devices and not for low-end devices targeted in this work. As discussed earlier, no prior hybrid RA architecture for low-end devices provides PoX.

5.3 Proof of Execution (PoX) Schemes

A *Proof of Execution* (PoX) is a scheme involving two parties: (1) a trusted verifier $\mathcal{V}$rf, and (2) an untrusted (potentially infected) remote prover $\mathcal{P}$rv. Informally, the goal of PoX is to allow $\mathcal{V}$rf to request execution of specific software $\mathcal{S}$ by $\mathcal{P}$rv. As part of PoX, $\mathcal{P}$rv must reply

Definition 9 (Proof of Execution (PoX) Scheme)**.**
A Proof of Execution (PoX) scheme is a tuple of algorithms [XRequest, XAtomicExec, XProve, XVerify] *performed between* $\mathcal{P}$rv *and* $\mathcal{V}$rf *where:*

1. XRequest$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}(\mathcal{S},\cdot)$*: is an algorithm executed by* $\mathcal{V}$rf *which takes as input some software* $\mathcal{S}$ *(consisting of a list of instructions* $\{s_1, s_2, ..., s_m\}$*).* $\mathcal{V}$rf *expects an honest* $\mathcal{P}$rv *to execute* $\mathcal{S}$*.* XRequest *generates a challenge* Chal*, and embeds it alongside* $\mathcal{S}$*, into an output request message asking* $\mathcal{P}$rv *to execute* $\mathcal{S}$*, and to prove that such execution took place.*

2. XAtomicExec$^{\mathcal{P}\text{rv}}(ER,\cdot)$*: an algorithm (with possible hardware-support) that takes as input some executable region* ER *in* $\mathcal{P}$rv*'s memory, containing a list of instructions* $\{i_1, i_2, ..., i_m\}$*.* XAtomicExec *runs on* $\mathcal{P}$rv *and is considered successful iff: (1) instructions in* ER *are executed from its first instruction,* i_1*, and end at its last instruction,* i_m*; (2)* ER *execution is atomic, i.e., if* E *is the sequence of instructions executed between* i_1 *and* i_m*, then* $\{e|e \in E\} \subseteq ER$*; and (3)* ER *execution flow is not altered by external events, i.e., MCU interrupts or DMA events. The* XAtomicExec *algorithm outputs result string* $\mathcal{O}$*. Note that* $\mathcal{O}$ *may be a default string (*$\perp$*) if* ER *execution does not result in any output.*

3. XProve$^{\mathcal{P}\text{rv}}(ER, \text{Chal}, \mathcal{O},\cdot)$*: an algorithm (with possible hardware-support) that takes as input some* ER*,* Chal *and* $\mathcal{O}$ *and is run by* $\mathcal{P}$rv *to output* $\mathcal{H}$*, i.e., a proof that* XRequest$^{\mathcal{V}\text{rf}\rightarrow\mathcal{P}\text{rv}}(\mathcal{S},\cdot)$ *and* XAtomicExec$^{\mathcal{P}\text{rv}}(ER,\cdot)$ *happened (in this sequence) and that* $\mathcal{O}$ *was produced by* XAtomicExec$^{\mathcal{P}\text{rv}}(ER,\cdot)$*.*

4. XVerify$^{\mathcal{P}\text{rv}\rightarrow\mathcal{V}\text{rf}}(\mathcal{H}, \mathcal{O}, \mathcal{S}, \text{Chal},\cdot)$*: an algorithm executed by* $\mathcal{V}$rf *with the following inputs: some* $\mathcal{S}$*,* Chal*,* $\mathcal{H}$ *and* $\mathcal{O}$*. The* XVerify *algorithm checks whether* $\mathcal{H}$ *is a valid proof of the execution of* $\mathcal{S}$ *(i.e., executed memory region* ER *corresponds to* $\mathcal{S}$*) on* $\mathcal{P}$rv *given the challenge* Chal*, and if* $\mathcal{O}$ *is an authentic output/result of such an execution. If both checks succeed,* XVerify *outputs 1, otherwise it outputs 0.*

   Remark: *In the parameters list,* $(\cdot)$ *denotes that additional parameters might be included, depending on the specific PoX construction.*

to $\mathcal{V}$rf with an authenticated unforgeable cryptographic proof ($\mathcal{H}$) that convinces $\mathcal{V}$rf that $\mathcal{P}$rv indeed executed $\mathcal{S}$. To accomplish this, verifying $\mathcal{H}$ must prove that: (1) $\mathcal{S}$ executed atomically, in its entirety, and that such execution occurred on $\mathcal{P}$rv (and not on some other device); and (2) any claimed result/output value of such execution, that is accepted as legitimate by $\mathcal{V}$rf, could not have been spoofed or modified. Also, the size and behavior (i.e., instructions) of $\mathcal{S}$, as well as the size of its output (if any), should be configurable and optionally specified by $\mathcal{V}$rf. In other words, PoX should provide proofs of execution for arbitrary software, along with corresponding authenticated outputs. (Note that $\mathcal{V}$rf is trusted to specify a functionally correct , bug-free, $\mathcal{S}$. See Section 5.3.1 for a discussion of vulnerabilities in $\mathcal{S}$ itself.) Definition 9 specifies PoX schemes in detail.

Definition 10 (PoX Security Game)**.**
– *Let* t_{req} *denote time when* $\mathcal{V}$rf *issues* $\mathcal{C}$hal $\leftarrow \mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}(\mathcal{S})$.
– *Let* t_{verif} *denote time when* $\mathcal{V}$rf *receives* $\mathcal{H}$ *and* $\mathcal{O}$ *back from* $\mathcal{P}$rv *in response to* $\mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}$.
– *Let* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$ *denote that* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER, \cdot)$*, such that* $ER \equiv \mathcal{S}$*, was invoked and completed within the time interval* $[t_{req}, t_{verif}]$.
– *Let* $\mathcal{O} \equiv \mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$ *denote that* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$ *produces output* $\mathcal{O}$*. Conversely,* $\mathcal{O} \not\equiv \mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$ *indicates* $\mathcal{O}$ *is not produced by* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$.
10.1 PoX Security Game (PoX-game): *Challenger plays the following game with* $\mathcal{A}$dv*:*

1. $\mathcal{A}$dv *is given full control over* $\mathcal{P}$rv *software state and oracle access to calls to the algorithms* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}$ *and* $\mathsf{XProve}^{\mathcal{P}\mathsf{rv}}$.
2. *At time* t_{req}*,* $\mathcal{A}$dv *is presented with software* $\mathcal{S}$ *and challenge* $\mathcal{C}$hal.
3. $\mathcal{A}$dv *wins in two cases:*
   (a) ***None or incomplete execution:*** $\mathcal{A}$dv *produces* ($\mathcal{H}_{\mathcal{A}\mathsf{dv}}$,$\mathcal{O}_{\mathcal{A}\mathsf{dv}}$)*, such that* $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{O}_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$*, without calling* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$.
   (b) ***Execution with tampered output:*** $\mathcal{A}$dv *calls* $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$ *and can produce* ($\mathcal{H}_{\mathcal{A}\mathsf{dv}}$,$\mathcal{O}_{\mathcal{A}\mathsf{dv}}$)*, such that* $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{O}_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$ *and* $\mathcal{O}_{\mathcal{A}\mathsf{dv}} \not\equiv \mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$

10.2 PoX Security Definition:
A PoX scheme is considered secure for security parameter $\mathtt{l}$ *if, for all PPT adversaries* $\mathcal{A}$dv*, there exists a negligible function* negl *such that:*

$$Pr[\mathcal{A}\mathsf{dv}, \textit{PoX-game}] \leq \mathsf{negl}(\mathtt{l})$$

We now justify the need to include atomic execution of $\mathcal{S}$ in the definition of PoX. On low-end MCUs, software typically runs on "bare metal" and, in most cases, there is no mechanism to enforce memory isolation between applications. Therefore, allowing $\mathcal{S}$ execution to be interrupted would permit other (potentially malicious) software running on $\mathcal{P}$rv to alter the behavior of $\mathcal{S}$. This might be done, for example, by an application that interrupts execution of $\mathcal{S}$ and changes intermediate computation results in $\mathcal{S}$ data memory, thus tampering with its output or control flow. Another example is an interrupt that resumes $\mathcal{S}$ at different instruction modifying $\mathcal{S}$ execution flow. Such actions could modify $\mathcal{S}$ behavior completely via Return Oriented Programming (ROP).

5.3.1 PoX Adversary Model & Security Definition

The adversary model considered by APEX is the same considered in Chapters 3 and 4. We review it below.

We consider an adversary $\mathcal{A}\mathsf{dv}$ that controls $\mathcal{P}\mathsf{rv}$'s entire software state, code, and data. $\mathcal{A}\mathsf{dv}$ can modify any writable memory and read any memory that is not explicitly protected by hardware-enforced access control rules. $\mathcal{A}\mathsf{dv}$ may also have full control over all Direct Memory Access (DMA) controllers of $\mathcal{P}\mathsf{rv}$. Recall that DMA allows a hardware controller to directly access main memory (e.g., RAM, flash or ROM) without going through the CPU.

We consider a PoX = (XRequest, XAtomicExec, XProve, XVerify) scheme to be secure if the aforementioned $\mathcal{A}\mathsf{dv}$ has only negligible probability of convincing $\mathcal{V}\mathsf{rf}$ that $\mathcal{S}$ executed successfully when, in reality, such execution did not take place, or was interrupted. In addition we require that, if execution of $\mathcal{S}$ occurs, $\mathcal{A}\mathsf{dv}$ can not tamper with, or influence, this execution's outputs. These notions are formalized by the security game in Definition 10.

We note that Definition 10 binds execution of $\mathcal{S}$ to the time between $\mathcal{V}\mathsf{rf}$ issuing the request and receiving the response. Therefore, if a PoX scheme is secure according to this definition, $\mathcal{V}\mathsf{rf}$ can be certain about freshness of the execution. In the same vein, the output produced by such execution is also guaranteed to be fresh. This timeliness property is important to avoid replays of previous valid executions; in fact, it is essential for safety-critical applications. See Section 5.6.3 for examples.

Correctness of the Executable: we stress that the purpose of PoX is to guarantee that $\mathcal{S}$, as specified by $\mathcal{V}\mathsf{rf}$, was executed. Similar to Trusted Execution Environments (TEEs) targeting high-end CPUs, such as Intel SGX, PoX schemes do not aim to check correctness and absence of implementation bugs in $\mathcal{S}$. As such, it is not concerned with run-time attacks that exploit bugs and vulnerabilities in $\mathcal{S}$ implementation itself, to change its expected behavior

(e.g., by executing $\mathcal{S}$ with inputs crafted to exploit $\mathcal{S}$ bugs and hijack its control flow). In particular, correctness (or termination) of $\mathcal{S}$ need **not** be assured by the low-end $\mathcal{P}$rv. Since $\mathcal{V}$rf is a more powerful device and knows $\mathcal{S}$, it has the ability (and more computational resources) to employ various vulnerability detection methods (e.g., fuzzing [33] or static analysis [38]) or even software formal verification (depending on the level of rigor desired) to avoid or detect implementation bugs in $\mathcal{S}$. This type of techniques can be performed offline before sending $\mathcal{S}$ to $\mathcal{P}$rv and the whole issue is orthogonal to the PoX functionality. We also note that, if $\mathcal{S}$ needs to be instrumented for PoX (see Section 5.4.1), it is important to ensure that this instrumentation does not introduce any bugs/vulnerabilities into $\mathcal{S}$.

Physical Attacks: physical and hardware-focused attacks are out of scope of this work. Specifically, we assume that $\mathcal{A}$dv can not modify code in ROM, induce hardware faults, or retrieve $\mathcal{P}$rv secrets via physical presence side-channels. Protection against such attacks is considered orthogonal and could be supported via standard physical security techniques [95]. This assumption is inline with other hybrid architectures [49, 40, 68, 22].

5.3.2 MCU Assumptions

We assume the same machine model as in Chapter 3 and make no additional assumptions. We review relevant assumptions throughout the rest of this section and then formalize them as an LTL machine model in Section 5.5.

Verification of the entire CPU is beyond the scope of this work. Therefore, we assume the CPU architecture strictly adheres to, and correctly implements, its specifications. In particular, our design and verification rely on the following simple axioms:

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

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

A3 – ***DMA:*** Whenever the DMA controller attempts to access the main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (logical zero).

A4 – ***MCU Reset:*** At the end of a successful reset routine, all registers (including PC) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine can not be modified. When a reset happens, the corresponding *reset* signal is set. The same signal is also set when the MCU initializes for the first time.

A5 – ***Interrupts:*** Whenever an interrupt occurs, the corresponding *irq* signal is set.

5.4 *APEX*: A Secure PoX Architecture

We now present *APEX*, a PoX architecture that satifies PoX-Security per Definition 10. This section first provides some intuition behind *APEX* design. All *APEX* properties are overviewed informally in this section and formalized in Section 5.5.

In the rest of this section we use the term "unprivileged software" to refer to any software other than SW-Att code from *VRASED*. $\mathcal{A}$dv is allowed to overwrite or bypass any "unprivileged software". Meanwhile, "trusted software" refers to *VRASED* implementation of SW-Att (see Chapter 3) which is formally verified and can not be modified by $\mathcal{A}$dv, since it

Definition 11 (Proof of Execution Protocol)**.** *APEX instantiates a* PoX = (XRequest, XAtomicExec, XProve, XVerify) *scheme behaving as follows:*

1. $\mathsf{XRequest}^{\mathcal{V}\mathsf{rf}\rightarrow\mathcal{P}\mathsf{rv}}(\mathcal{S}, ER_{min}, ER_{max}, OR_{min}, OR_{max})$*: includes a set of configuration parameters* ER_{min}*,* ER_{max}*,* OR_{min}*,* OR_{max}*. The Executable Range (ER) is a contiguous memory block in which* $\mathcal{S}$ *is to be installed:* $ER = [ER_{min}, ER_{max}]$*. Similarly, the Output Range (OR) is also configurable and defined by* $\mathcal{V}\mathsf{rf}$*'s request as* $OR = [OR_{min}, OR_{max}]$*. If* $\mathcal{S}$ *does not produce any output* $OR_{min} = OR_{max} = \perp$*.* $\mathcal{S}$ *is the software to be installed in ER and executed. If* $\mathcal{S}$ *is unspecified (*$\mathcal{S} = \perp$*) the protocol will execute whatever code was pre-installed on ER on* $\mathcal{P}\mathsf{rv}$*, i.e.,* $\mathcal{V}\mathsf{rf}$ *is not required to provide* $\mathcal{S}$ *in every request, only when it wants to update ER contents before executing it. If the code for* $\mathcal{S}$ *is sent by* $\mathcal{V}\mathsf{rf}$*, untrusted auxiliary software in* $\mathcal{P}\mathsf{rv}$ *is responsible for copying* $\mathcal{S}$ *into ER.* $\mathcal{P}\mathsf{rv}$ *also receives a random l-bit challenge* $\mathcal{C}\mathsf{hal}$ *(*$|\mathcal{C}\mathsf{hal}| = l$*) as part of the request, where l is the security parameter.*

2. $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER, OR, METADATA)$*: This algorithm starts with unprivileged auxiliary software writing the values of:* ER_{min}*,* ER_{max}*,* OR_{min}*,* OR_{max} *and* $\mathcal{C}\mathsf{hal}$ *to a special pre-defined memory region denoted by* $METADATA$*. APEX verified hardware enforces immutability, atomic execution and access control rules according to the values stored in* $METADATA$*; details are described in Section 5.4.1. Finally, it begins execution of* $\mathcal{S}$ *by setting the program counter to the value of* ER_{min}*.*

3. $\mathsf{XProve}^{\mathcal{P}\mathsf{rv}}(ER, \mathcal{C}\mathsf{hal}, OR)$*: produces proof of execution* $\mathcal{H}$*.* $\mathcal{H}$ *allows* $\mathcal{V}\mathsf{rf}$ *to decide whether: (1) code contained in ER actually executed; (2) ER contained specified (expected)* $\mathcal{S}$ *code during execution; (3) this execution is fresh, i.e., performed after the most recent* XRequest*; and (4) claimed output in OR is indeed produced by this execution. As mentioned earlier, APEX uses VRASED* RA *architecture to compute* $\mathcal{H}$ *by attesting at least the executable, along with its output, and corresponding execution metadata. More formally:*

$$\mathcal{H} = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), ER, OR, METADATA, ...) \tag{5.1}$$

$METADATA$ also contains the $EXEC$ flag that is ***read-only to all software running in*** $\mathcal{P}\mathsf{rv}$ ***and can only be written to by*** *APEX* ***formally verified hardware****. This hardware monitors execution and sets* $EXEC = 1$ *only if ER executed successfully (*XAtomicExec*) and memory regions of* $METADATA$*, ER, and OR were not modified between the end of ER execution and the computation of* $\mathcal{H}$*. The reasons for these requirements are detailed in Section 5.4.2. If any malware residing on* $\mathcal{P}\mathsf{rv}$ *attempts to violate any of these properties APEX verified hardware (provably) sets EXEC to zero. After computing* $\mathcal{H}$*,* $\mathcal{P}\mathsf{rv}$ *returns it and contents of OR (*$\mathcal{O}$*) produced by ER execution to* $\mathcal{V}\mathsf{rf}$*.*

4. $\mathsf{XVerify}^{\mathcal{P}\mathsf{rv}\rightarrow\mathcal{V}\mathsf{rf}}(\mathcal{H}, \mathcal{O}, \mathcal{S}, METADATA_{\mathcal{V}\mathsf{rf}})$ *: Upon receiving* $\mathcal{H}$ *and* $\mathcal{O}$*,* $\mathcal{V}\mathsf{rf}$ *checks whether* $\mathcal{H}$ *is produced by a legitimate execution of* $\mathcal{S}$ *and reflects parameters specified in* XRequest*, i.e.,* $METADATA_{\mathcal{V}\mathsf{rf}} = \mathcal{C}\mathsf{hal}||OR_{min}||OR_{max}||ER_{min}||ER_{max}||EXEC = 1$*. This way,* $\mathcal{V}\mathsf{rf}$ *concludes that* $\mathcal{S}$ *successfully executed on* $\mathcal{P}\mathsf{rv}$ *and produced output* $\mathcal{O}$ *if:*

$$\mathcal{H} \equiv HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}_{\mathcal{V}\mathsf{rf}}), \mathcal{S}, \mathcal{O}, METADATA_{\mathcal{V}\mathsf{rf}}, ...) \tag{5.2}$$

is stored in ROM. *APEX* is designed such that no changes to SW-Att are required. Therefore, both functionalities (RA and PoX, i.e., *VRASED* and *APEX*) can co-exist on the same device

Table 5.1: Summary of *APEX*-relevant notation

PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
irq	Signal that indicates if an interrupt is happening
CR	Memory region where `SW-Att` is stored: $CR = [CR_{min}, CR_{max}]$
MR	(MAC Region) Memory region in which `SW-Att` computation result is written: $MR = [MR_{min}, MR_{max}]$. The same region is used to pass the attestation challenge as input to `SW-Att`
AR	(Attested Region) Memory region to be attested. Can be fixed/predefined or specified in an authenticated request from $\mathcal{V}$rf: $AR = [AR_{min}, AR_{max}]$
KR	(Key Region) Memory region that stores $\mathcal{K}$
XS	(Exclusive Stack Region) Exclusive memory region that contains `SW-Att` stack and can be only accessed by `SW-Att`
$reset$	A 1-bit signal that reboots/resets the MCU when set to logical 1
ER	(Execution Region) Memory region that stores an executable to be executed: $ER = [ER_{min}, ER_{max}]$
OR	(Output Region) Memory region that stores execution output: $OR = [OR_{min}, OR_{max}]$
$EXEC$	1-bit execution flag indicating whether a successful execution has happened
$METADATA$	Memory region containing *APEX* metadata

without interfering with each other (i.e., it is still possible to attest memory without proving its execution, if that is the desired behavior).

Notation is summarized in Table 5.1.

5.4.1 Protocol and Architecture

APEX implements a secure `PoX` scheme conforming to Definition 11. The steps in *APEX* workflow are illustrated in Figure 5.1. The main idea is to first execute code contained in ER. Then, at some later time, *APEX* invokes *VRASED* to attest the code in ER and include, in the attestation result, additional information that allows $\mathcal{V}$rf to verify that ER code actually executed. If ER execution produces an output (e.g., $\mathcal{P}$rv is a sensor running $\mathcal{S}$ to obtain some physical/ambient quantity), authenticity and integrity of this output can

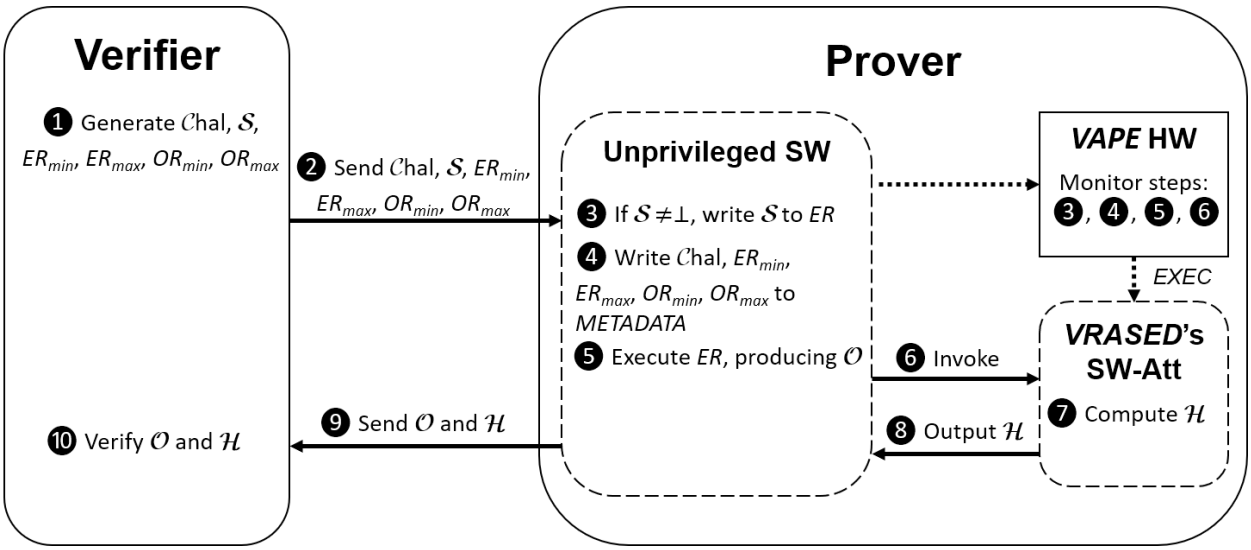


Figure 5.1: Overview of *APEX* workflow

also be verified. That is achieved by including the $EXEC$ flag among inputs to HMAC computed as part of *VRASED* RA. The value of this flag is controlled by *APEX* formally verified hardware and **its memory can not be written by any software running on** $\mathcal{P}$rv. *APEX* hardware module runs in parallel with the MCU, monitoring its behavior and deciding the value of $EXEC$ accordingly.

Figure 5.2 depicts *APEX* architecture. In addition to *VRASED* hardware that provides secure RA by monitoring a set of CPU signals (see Chapter 3), *APEX* monitors values stored in the dedicated physical memory region called $METADATA$. It contains addresses/pointers to memory boundaries of ER (i.e., ER_{min} and ER_{max}) and memory boundaries of expected output: OR_{min} and OR_{max}. These addresses are sent by $\mathcal{V}$rf as part of XRequest, and are configurable at run-time. The code $\mathcal{S}$ to be stored in ER is optionally sent by $\mathcal{V}$rf. Sending $\mathcal{S}$ is optional because $\mathcal{S}$ might be pre-installed on $\mathcal{P}$rv. In that case the proof of execution will allow $\mathcal{V}$rf to conclude that the pre-installed $\mathcal{S}$ was unmodified and executed.

$METADATA$ includes the $EXEC$ flag, which is initialized to 0 and only changes from 0 to 1 (by *APEX* hardware) when ER execution starts, i.e., when the $PC = ER_{min}$. Afterwards, any violation of *APEX* security properties (detailed in Section 5.4.2) immediately changes

$EXEC$ back to 0. After a violation, the only way to set $EXEC$ back to 1 is to re-start execution of ER from the very beginning, i.e., with PC=ER_{min}. In other words, $APEX$ verified hardware makes sure that $EXEC$ value covered by the HMAC result (represented by $\mathcal{H}$) is 1, if and only if ER code executed successfully. As mentioned earlier, we consider an execution to be successful if it runs atomically (i.e., from beginning to end and interrupted).

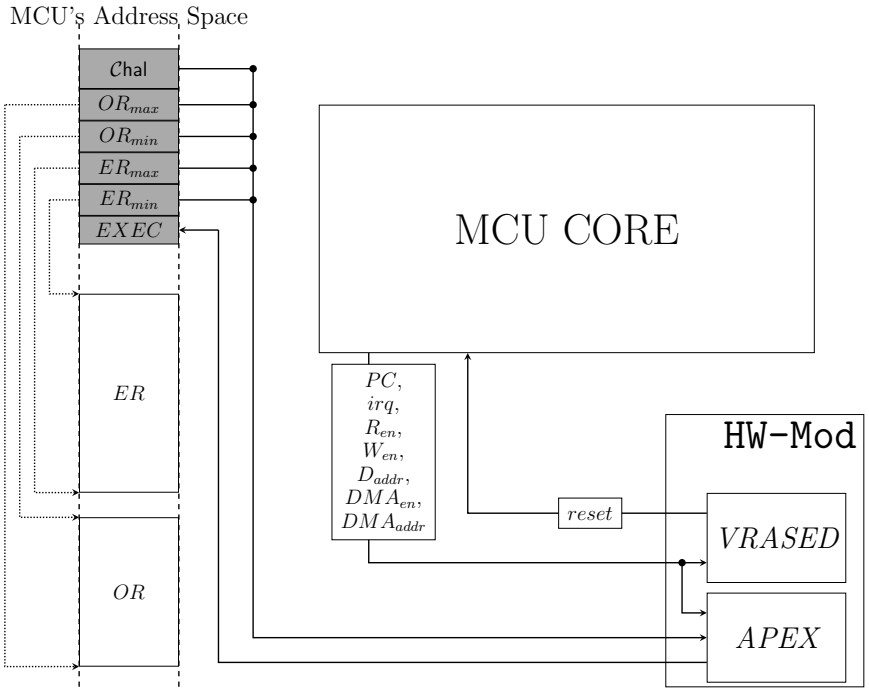


Figure 5.2: `HW-Mod` composed of $APEX$ and $VRASED$ hardware modules. Shaded area represents $APEX\ METADATA$.

In addition to $EXEC$, HMAC covers a set of parameters (in $METADATA$ memory region) that allows $\mathcal{V}$rf to check whether executed software was indeed located in $ER = [ER_{min}, ER_{max}]$. If any output is expected, $\mathcal{V}$rf specifies a memory range $OR = [OR_{min}, OR_{max}]$ for storing output. Contents of OR are also covered by HMAC, allowing $\mathcal{V}$rf to verify authenticity of the output.

Remark: Our notion of successful execution requires $\mathcal{S}$ to have a single exit point – ER_{max}. Any self-contained code with multiple legal exits can be trivially instrumented to have a single exit point by *replacing each exit instruction with a jump to the unified exit point* ER_{max}.

This notion also requires $\mathcal{S}$ to run atomically. Since this constraint might be undesirable for some real-time systems, we discuss how to relax it in Section 5.7. In addition, $\mathcal{V}$rf is responsible for defining OR memory region according to $\mathcal{S}$ behavior. OR should be large enough to fit all output produced by $\mathcal{S}$ and OR boundaries should correspond to addresses where $\mathcal{S}$ writes its output values to be sent to $\mathcal{V}$rf. To ensure freshness of OR content, $\mathcal{V}$rf may enforce ER to clear OR contents as the first step in its execution. This may be necessary if not all ER execution paths overwrite OR entirely.

5.4.2 *APEX* Sub-Properties at a High-Level

We now describe sub-properties enforced by *APEX*. Section 5.5 formalizes them in LTL and provides a single end-to-end definition of *APEX* correctness. This end-to-end correctness notion is provably implied by the composition of all sub-properties. Sub-properties fall into two major groups: *Execution Protection* and *Metadata Protection*. A violation of any of these properties implies one or more of:

- Code in ER was not executed atomically and in its entirety;
- Output in OR was not produced by ER execution;
- Code in ER was not executed in a timely manner, i.e., after receiving the latest XRequest.

Whenever *APEX* detects a violation, $EXEC$ is set to 0. Since $EXEC$ is included among inputs to the computation of HMAC (conveyed in $\mathcal{P}$rv's response), it will be interpreted by $\mathcal{V}$rf as failure to prove execution of code in ER.

Remark: We emphasize that properties discussed below are specific to PoX and are required **in addition** to *VRASED* verified properties. They should not be viewed as replacements for any of *VRASED* properties that are used to enforce RA security.

Execution Protection:

EP1 – ***Ephemeral Immutability:*** Code in ER can not be modified from the start of its execution until the end of `SW-Att` computation in XProve routine. This property is necessary to ensure that the attestation result reflects the code that executed. Lack of this property would allow $\mathcal{A}$dv to execute some other code $ER_{\mathcal{A}\mathsf{dv}}$, overwrite it with expected ER and finally call XProve. This would result in a valid proof of execution of ER even though $ER_{\mathcal{A}\mathsf{dv}}$ was executed instead.

EP2 – ***Ephemeral Atomicity:*** ER execution is only considered successful if ER runs starting from ER_{min} until ER_{max} atomically, i.e., without any interruption. This property conforms with XAtomicExec routine in Definition 9 and with the notion of successful execution in the context of our work. As discussed in Section 5.3, ER must run atomically to prevent malware residing on $\mathcal{P}$rv from interrupting ER execution and resuming it at a different instruction, or modifying intermediate execution results in data memory. Without this property, Return-Oriented Programming (ROP) and similar attacks on ER could change its behavior completely and unpredictably, making any proof of execution (and corresponding output) useless.

EP3 – ***Output Protection:*** Similar to **EP1**, *APEX* must ensure that OR is unmodified from the time after ER code execution is finished until completion of HMAC computation in XProve. Lack of this property would allow $\mathcal{A}$dv to overwrite OR and successfully spoof OR produced by ER, thus convincing $\mathcal{V}$rf that execution produced some forged output.

Metadata Protection:

MP1 - ***Executable/Output (ER/OR) Boundaries:*** *APEX* hardware ensures properties **EP1**, **EP2**, and **EP3** according to values: ER_{min}, ER_{max}, OR_{min}, OR_{max}. These

values are configurable by $\mathcal{V}$rf based on application needs. They are written into metadata-dedicated physical addresses in $\mathcal{P}$rv memory before ER execution. Therefore, once ER execution starts, $APEX$ hardware must ensure that such values remain unchanged until XProve completes. Otherwise, $\mathcal{A}$dv could generate valid attestation results, by attesting $[ER_{min}, ER_{max}]$, while, in fact, having executed code in a different region: $[ER_{min}^{\mathcal{A}\mathsf{dv}}, ER_{max}^{\mathcal{A}\mathsf{dv}}]$.

MP2 - *Response Protection:* The appropriate response to $\mathcal{V}$rf's challenge must be unforgeable and non-invertible. Therefore, in the XProve routine, $\mathcal{K}$ must never be leaked (with non-negligible probability) and HMAC implementation must be functionally correct, i.e., adhere to its cryptographic specification. Moreover, contents of memory being attested must not change during HMAC computation. We rely on *VRASED* to ensure these properties. Also, to ensure trustworthiness of the response, $APEX$ guarantees that no software in $\mathcal{P}$rv can ever modify $EXEC$ flag and that, once $EXEC = 0$, it can only become 1 if ER execution re-starts afresh.

MP3 - *Challenge Temporal Consistency:* $APEX$ must ensure that $\mathcal{C}$hal can not be modified between ER execution and HMAC computation in XProve. Without this property, the following attack is possible: (1) $\mathcal{P}$rv-resident malware executes ER properly (i.e., by not violating **EP1-EP3** and **MP1-MP2**), resulting in $EXEC = 1$ after execution stops, and (2) at a later time, malware receives $\mathcal{C}$hal from $\mathcal{V}$rf and simply calls XProve on this $\mathcal{C}$hal without executing ER. As a result, malware would acquire a valid proof of execution (since $EXEC$ remains 1 when the proof is generated) even though no ER execution occurred before $\mathcal{C}$hal was received. Such attacks are prevented by setting $EXEC = 0$ whenever the memory region storing $\mathcal{C}$hal is modified.

5.5 Formal Specification & Verified Implementation

Our formal verification approach starts by formalizing *APEX* sub-properties Linear Temporal Logic (LTL) to define invariants that must hold throughout the MCU operation. We then use a theorem prover [47] to write a computer-aided proof that the conjunction of the LTL sub-properties imply an end-to-end formal definition for the guarantee expected from *APEX* hardware. *APEX* correctness, when properly composed with *VRASED* guarantees, yields a `PoX` scheme secure according to Definition 10. This is proved by showing that, if the composition between the two is implemented as described in Definition 11, *VRASED* security can be reduced to *APEX* security.

APEX hardware module is composed of several sub-modules written in Verilog Hardware Description Language (HDL). Each sub-module is responsible for enforcing a set of LTL sub-properties and is described as an FSM in Verilog at Register Transfer Level (RTL). Individual sub-modules are combined into a single Verilog design. The resulting composition is converted to the SMV model checking language using the automatic translation tool Verilog2SMV [65]. The resulting SMV is simultaneously verified against all LTL specifications, using the model checker NuSMV[35], to prove that the final implementation *APEX* complies with all necessary properties.

5.5.1 Machine Model

Definition 12 models, in LTL, the behavior of low-end MCUs considered in this work. It consists of a subset of the machine model introduced by *VRASED*. Nonetheless, this subset models all MCU behavior relevant for stating and verifying correctness of *APEX* implementation.

`Modify_Mem` models that a given memory address can be modified by a CPU instruction or

Definition 12. *Machine Model (subset)*

1. *Modify_Mem(i)* $\rightarrow (W_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$
2. *Interrupt* $\rightarrow$ *irq*
3. MR, CR, AR, KR, XS, *and* $METADATA$ *are non-overlapping memory regions*

by a DMA access. In the former, W_{en} signal must be set and D_{addr} must contain the target memory address. In the latter, DMA_{en} signal must be set and DMA_{addr} must contain the target DMA address. The requirements for *reading from* a memory address are similar, except that instead of W_{en}, R_{en} must be on. We do not explicitly state this behavior since it is not used in *APEX* proofs. For the same reason, modeling the effects of instructions that only modify register values (e.g., ALU operations, such as *add* and *mul*) is also unnecessary. The machine model also captures the fact that, when an interrupt happens during execution, the *irq* signal in MCU hardware is set to 1.

With respect to memory layout, the model states that MR, CR, AR, KR, XS, and $METADATA$ are disjoint memory regions. The first five memory regions are defined in *VRASED*. As shown in Figure 5.2, $METADATA$ is a fixed memory region used by *APEX* to store information about software execution status.

5.5.2 Security & Implementation Correctness

We use a two-part strategy to prove that *APEX* is a secure `PoX` architecture, according to Definition 10:

[A]: We show that properties **EP1-EP3** and **MP1-MP3**, discussed in Section 5.4.2 and formally specified next in Section 5.5.3, are sufficient to guarantee that $EXEC$ flag is 1 iff $\mathcal{S}$ indeed executed on $\mathcal{P}$rv. To show this, we compose a computer proof using SPOT LTL proof assistant [47].

[B]: We use cryptographic reduction proofs to show that, as long as part **A** holds, *VRASED*

security can be reduced to *APEX* PoX security from Definition 10. In turn, HMAC existential unforgeability can be reduced to *VRASED* security [40]. Therefore, both *APEX* and *VRASED* rely on the assumption that HMAC is a secure MAC.

Definition 13. *Formal specification of APEX correctness.*

$$\{ \\ PC = ER_{min} \ \wedge \ [(PC \in ER \wedge \neg Interrupt \wedge \neg reset \wedge \neg DMA_{en}) \quad \boldsymbol{U} \quad PC = ER_{max}] \quad \wedge \\ [(\neg \ Modify\_Mem(ER) \wedge \neg \ Modify\_Mem(METADATA) \wedge (PC \in ER \vee \neg \ Modify\_Mem(OR))) \quad \boldsymbol{U} \quad PC = CR_{min}] \\ \} \quad \boldsymbol{B} \quad \{EXEC \wedge PC \in CR\} \tag{5.3}$$

Definition 14. *Sub-Properties needed for Secure Proofs of Execution in LTL.*
Ephemeral Immutability:

$$\mathbf{G}: \ \{[W_{en} \wedge (D_{addr} \in ER)] \vee [DMA_{en} \wedge (DMA_{addr} \in ER)] \rightarrow \neg EXEC\} \tag{5.4}$$

Ephemeral Atomicity:

$$\mathbf{G}: \ \{(PC \in ER) \wedge \neg(\mathbf{X}(PC) \in ER) \rightarrow PC = ER_{max} \vee \neg\mathbf{X}(EXEC) \ \} \tag{5.5}$$

$$\mathbf{G}: \ \{\neg(PC \in ER) \wedge (\mathbf{X}(PC) \in ER) \rightarrow \mathbf{X}(PC) = ER_{min} \vee \neg\mathbf{X}(EXEC)\} \tag{5.6}$$

$$\mathbf{G}: \ \{(PC \in ER) \wedge irq \rightarrow \neg EXEC\} \tag{5.7}$$

Output Protection:

$$\mathbf{G}: \ \{[\neg(PC \in ER) \wedge (W_{en} \wedge D_{addr} \in OR)] \vee (DMA_{en} \wedge DMA_{addr} \in OR) \vee (PC \in ER \wedge DMA_{en}) \rightarrow \neg EXEC\} \tag{5.8}$$

Executable/Output (ER/OR) Boundaries & Challenge Temporal Consistency:

$$\mathbf{G}: \ \{ER_{min} > ER_{max} \vee OR_{min} > OR_{max} \rightarrow \neg EXEC\} \tag{5.9}$$

$$\mathbf{G}: \ \{ER_{min} \leq CR_{max} \vee ER_{max} > CR_{max} \rightarrow \neg EXEC\} \tag{5.10}$$

$$\mathbf{G}: \ \{[W_{en} \wedge (D_{addr} \in METADATA)] \vee [DMA_{en} \wedge (DMA_{addr} \in METADATA)] \rightarrow \neg EXEC\} \tag{5.11}$$

Remark: *Note that* $\mathsf{Chal}_{mem} \in METADATA$.

Response Protection:

$$\mathbf{G}: \ \{\neg EXEC \wedge \mathbf{X}(EXEC) \rightarrow \mathbf{X}(PC = ER_{min})\} \tag{5.12}$$

$$\mathbf{G}: \ \{reset \rightarrow \neg EXEC\} \tag{5.13}$$

In the rest of this section, we convey the intuition behind both of these steps. Proof details

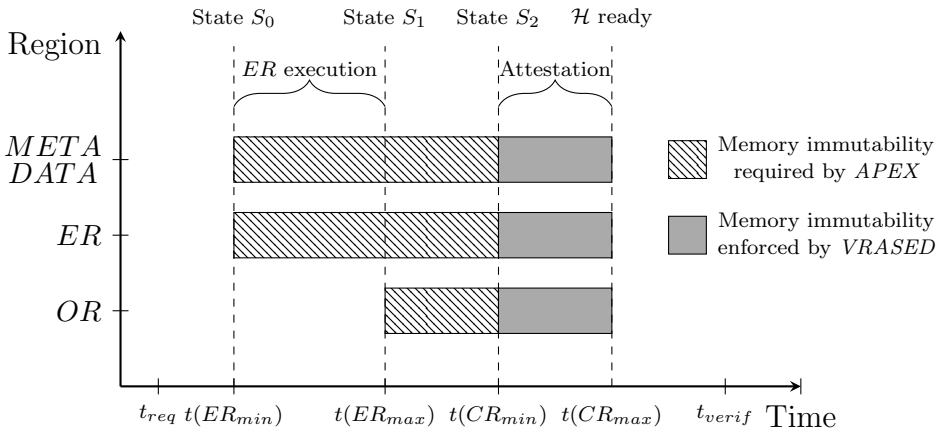


Figure 5.3: Illustration of time intervals that each memory region must remain unchanged in order to produce a valid $\mathcal{H}$ ($EXEC = 1$). $t(X)$ denotes the time when $PC = X$.

are in Appendix 5.10.

The goal of part **A** is to show that *APEX* sub-properties imply Definition 13. LTL specification in Definition 13 captures the conditions that must hold in order for $EXEC$ to be set to 1 during execution of XProve, enabling generation of a valid proof of execution. This specification ensures that, in order to have $EXEC = 1$ during execution of XProve (i.e, for $[EXEC \wedge PC \in CR]$ to hold), at least once **before such time** the following must have happened:

1. The system reached state S_0 where software stored in ER started executing from its first instruction ($PC = ER_{min}$).
2. The system eventually reached a state S_1 when ER finished executing ($PC = ER_{max}$). In the interval between S_0 and S_1 PC kept executing instructions within ER, there were no interrupts, no resets, and DMA remained inactive.
3. The system eventually reached a state S_2 when XProve started executing ($PC = CR_{min}$). In the interval between S_0 and S_2, $METADATA$ and ER regions were not modified.
4. In the interval between S_0 and S_2, OR region was only modified by ER execution, i.e., $PC \in ER \vee \neg$ Modify_Mem(OR).

Figure 5.3 shows the time windows wherein each memory region must not change during *APEX* PoX as implied by *APEX* correctness (Definition 13). Violating any of these conditions will cause $EXEC$ have value 0 during XProve computation. Consequently, any violation will result in $\mathcal{V}$rf rejecting the proof of execution since it will not conform to the expected value of $\mathcal{H}$, per Equation 5.2 in Definition 11.

The intuition behind the cryptographic reduction (part **B**) is that computing $\mathcal{H}$ involves simply invoking *VRASED* SW-Att with $MR = \mathcal{C}$hal, $ER \in AR$, $OR \in AR$, and $METADATA \in AR$. Therefore, a successful forgery of $APEX\mathcal{H}$ implies breaking *VRASED* security. Since $\mathcal{H}$ always includes the value of $EXEC$, this implies that *APEX* is PoX-secure (Definition 10). The complete reduction is presented in Appendix 5.10.

5.5.3 *APEX* Sub-Properties in LTL

We formalize the necessary sub-properties enforced by *APEX* as LTL specifications 5.4–5.13 in Definition 14. We describe how they map to high-level notions **EP1-EP3** and **MP1-MP3** discussed in Section 5.4.2. Appendix 5.10 discusses a computer proof that the conjunction of this set of properties is sufficient to satisfy a formal definition of *APEX* correctness from Definition 13.

LTL 5.4 enforces **EP1 – Ephemeral immutability** by making sure that whenever ER memory region is written by either CPU or DMA, $EXEC$ is immediately set to logical 0 (false).

EP2 – Ephemeral Atomicity is enforced by a set of three LTL specifications. LTL 5.5 enforces that the only way for ER execution to terminate, without setting $EXEC$ to logical 0, is through its last instruction: $PC = ER_{max}$. This is specified by checking the relation between current and next PC values using LTL ne**X**t operator. In particular, if current PC

value is within ER, and next PC value is outside `SW-Att` region, then either current PC value is the address of ER_{max}, or $EXEC$ is set to 0 in the next cycle. Also, LTL 5.6 enforces that the only way for PC to enter ER is through the very first instruction: ER_{min}. This prevents ER execution from starting at some point in the middle of ER, thus making sure that ER always executes in its entirety. Finally, LTL 5.7 enforces that $EXEC$ is set to zero if an interrupt happens in the middle of ER execution. Even though LTLs 5.5 and 5.6 already enforce that PC can not change to anywhere outside ER, interrupts could be programmed to return to an arbitrary instruction within ER. Although this would not violate LTLs 5.5 and 5.6, it would still modify ER behavior. Therefore, LTL 5.7 is needed to prevent that.

EP3 – Output Protection is enforced by LTL 5.8: (1) DMA controller does not write into OR; (2) CPU can only modify OR when executing instructions within ER; and 3) DMA can not be active during ER execution; otherwise, a compromised DMA could change intermediate results of ER computation in data memory, potentially modifying ER behavior.

Similar to **EP3**, **MP1 – Executable/Output Boundaries** and **MP3 – Challenge Temporal Consistency** are enforced by LTL 5.11. Since $\mathcal{C}$hal as well as ER_{min}, ER_{max}, OR_{min}, and OR_{max} are all stored in $METADATA$, it suffices to ensure that $EXEC$ is set to 0 whenever this region is modified. Also, LTL 5.9 enforces that $EXEC$ is only set to 1 if ER and OR are configured (by $METADATA$ values ER_{min}, ER_{max}, OR_{min}, OR_{max}) as valid memory regions.

Finally, LTLs 5.12, and 5.13 (in addition to *VRASED* verified RA architecture) are responsible for ensuring **MP2- Response Protection** by making sure that $EXEC$ always reflects what is intended by *APEX* hardware. LTL 5.8 specifies that the only way to change $EXEC$ from 0 to 1 is by starting ER execution over. Finally, LTL 5.13 states that, whenever a reset happens (this also includes the system initial booting state) and execution is initialized, the initial value of $EXEC$ is 0. To conclude, recall that $EXEC$ is read-only to all software running on $\mathcal{P}$rv. Therefore, malware can not change it directly.

	Hardware		Reserved	Verification			
	Reg	LUT	RAM (bytes)	# LTL Invariants	Verified Verilog LoC	Time (s)	Mem (MB)
OpenMSP430 [56]	691	1904	0	-	-	-	-
VRASED [40]	721	1964	2332	10	481	0.4	13.6
APEX + *VRASED*	735	2206	2341	20	1385	183.6	280.3

Table 5.2: Evaluation results.

APEX is designed as a set of seven hardware sub-modules, each verified to enforce a subset of properties discussed in this section. Examples of implementation of verified sub-modules as FSMs are discussed in Appendix 5.9.

5.6 Implementation & Evaluation

APEX is implemented on OpenMSP430 [56] core as illustrated in Figure 5.2. In addition to *APEX* and *VRASED* modules in `HW-Mod`, the prototype includes a peripheral module responsible for storing and maintaining $METADATA$. As a peripheral, contents of $METADATA$ can be accessed in a pre-defined memory address via standard peripheral memory access. We also ensure that $EXEC$ (located inside $METADATA$) is unmodifiable in software by removing software-write wires in hardware. Finally, as a proof of concept, we use Xilinx Vivado to synthesize an RTL description of the modified `HW-Mod` and deploy it on the Artix-7 FPGA class.

5.6.1 Evaluation Results

Hardware & Memory Overhead. Table 5.2 reports *APEX* hardware overhead as compared to plain OpenMSP430 [56] and *VRASED* [40]. Similar to the related work [40, 116, 45, 44], we consider the hardware overhead in terms of additional LUTs and registers. The increase in the number of LUTs can be used as an estimate of the additional chip cost and size required for combinatorial logic, while the number of registers offers an estimate on the

memory overhead required by the sequential logic in *APEX* FSMs. *APEX* hardware overhead is small compared to the baseline *VRASED*; it requires 2% and 12% additional registers and LUTs, respectively. In absolute numbers, it adds 44 registers and 302 Look-Up Tables (LUTs) to the underlying MCU. In terms of memory, *APEX* needs 9 extra bytes of `RAM` for storing $METADATA$. This overhead corresponds to 0.01% of MSP430 16-bit address space.

Run-time. We do not observe any software runtime overhead on the *APEX*-enabled $\mathcal{P}$rv since *APEX* does not introduce new instructions or modifications to the MSP430 ISA. *APEX* hardware runs in parallel with the original MSP430 CPU. Run-time to produce a proof of $\mathcal{S}$ execution includes: (1) time to execute $\mathcal{S}$ (XAtomicExec), and (2) time to compute an attestation token (XProve). The former only depends on $\mathcal{S}$ behavior itself (e.g., `SW-Att` can be a small sequence of instructions or have long loops). As mentioned earlier, *APEX* does not affect $\mathcal{S}$ run time. XProve run-time is linear in the size of $ER + OR$. In the worst-case scenario where these regions occupy the entire program 8kBytes memory, XProve takes around 900ms on an 8MHz device.

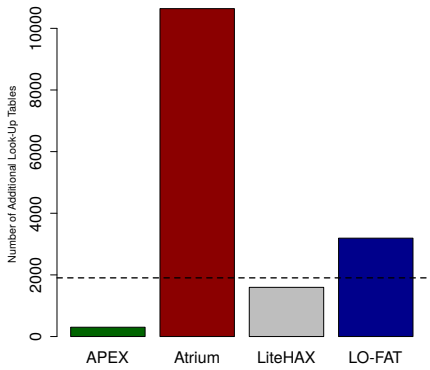
Verification Cost. We verify *APEX* on an Ubuntu 16.04 machine running at 3.40GHz. Results are shown in Table 5.2. *APEX* verification requires checking 10 extra invariants (shown in Definition 14) in addition to existing *VRASED* invariants. It also consumes significantly higher run-time and memory usage than *VRASED* verification. This is because additional invariants introduce five additional variables (ER_{min}, ER_{max}, OR_{min}, OR_{max} and $EXEC$), resulting in an exponential increase in complexity of the model checking process. Nonetheless, the overall verification process is still reasonable for a commodity desktop – it takes around 3 minutes and consumes 280MB of memory.

5.6.2 Comparison with CFA

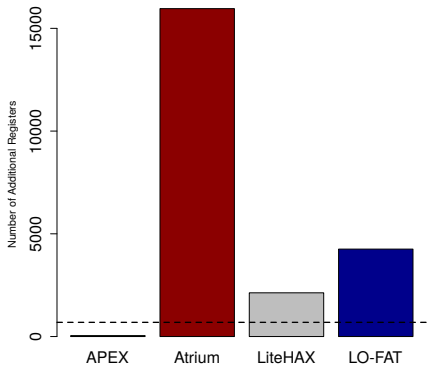
To the best of our knowledge, *APEX* is the first `PoX` architecture for low-end devices. Therefore, no other prior architecture is directly comparable. However, to provide a (performance and overhead) point of reference and a comparison, we contrast *APEX* overhead with state-of-the-art CFA architectures. As discussed in Section 5.2, even though CFA does not yield `PoX` with authenticated outputs, we consider it to be the closest-related service, since it reports on the exact execution path of a program.

We consider three recent CFA architectures: Atrium [116], LiteHAX [44], and LO-FAT [45]. Figure 5.4.a compares them to *APEX* in terms of number of additional LUTs. In this figure, the black dashed line represents the total cost of the MSP430 MCU: 1904 LUTs. Figure 5.4.b presents a similar comparison for the amount of additional registers required by these architectures. In this case, the total cost of the MSP430 MCU core is of 691 registers. Finally, Figure 5.4.c presents the amount of dedicated RAM required by these architectures (*APEX* dedicated RAM corresponds to the exclusive access stack implemented by *VRASED*).

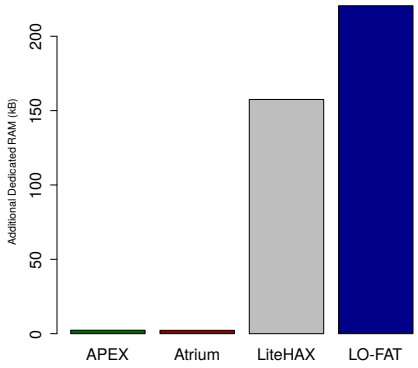
As expected, *APEX* incurs much lower overhead. The cheapest CFA architecture, LiteHAX, would entail an overhead of nearly 100% LUTs and 300% registers. In addition, LiteHAX would require 150 kB of dedicated RAM. This amount exceeds entire addressable memory (64 kB) of 16-bit processors, such as MSP430. Results support our claim that CFA is not applicable to this class of low-end devices. Meanwhile, *APEX* needs a total of 12% additional LUTs and 2% additional registers. *VRASED* requires about 2 kB of reserved RAM, which is not increased by *APEX*.



(a) % extra HW overhead: # Look-Up Tables



(b) % extra HW overhead: # Registers



(c) Dedicated RAM

Figure 5.4: Overhead comparison between *APEX* and CFA architectures. Dashed lines represent total hardware cost of MSP430.

5.6.3 Proof of Concept: Authenticated Sensing and Actuation

As discussed in Section 5.1 an important functionality attainable with `PoX` is authenticated sensing/actuation. In this section, we demonstrate how to use *APEX* to build sensors and actuators that "can not lie".

As a running example we use a fire sensor: a safety-critical low-end embedded device commonly present in households and workplaces. It consists of an MCU equipped with analog hardware for measuring physical quantities, e.g., temperature, humidity, and CO_2 level. It

is also usually equipped with actuation-capable analog hardware, such as a buzzer. Analog hardware components are directly connected to MCU General Purpose Input/Output (GPIO) ports. GPIO ports are physical wires directly mapped to fixed memory locations in MCU memory. Therefore, software running on the MCU can read physical quantities directly from GPIO memory.

In this example, we consider that MCU software periodically reads these values and transmits them to a remote safety authority, e.g., a fire department, which then decides whether to take action. The MCU also triggers the buzzer actuator whenever sensed values indicate a fire. Given the safety-critical nature of this application, the safety authority must be assured that reported values are authentic and were produced by execution of expected software. Otherwise, malware could spoof such values (e.g., by not reading them from the proper GPIO port or simply modifying them). `PoX` guarantees that reported values were read from the correct GPIO port (since the memory address is specified by instructions in the ER executable), and produced output (stored in OR) was indeed generated by execution of ER and was unmodified thereafter. Thus, upon receiving sensed values accompanied by a `PoX`, the safety authority is assured that the reported sensed value can be trusted.

As a proof of concept, we use *APEX* to implement a simple fire sensor that operates with temperature and humidity quantities. It communicates with a remote $\mathcal{V}$rf (e.g., the fire department) using a low-power ZigBee radio[1] typically used by low-end CPS/IoT devices. Temperature and humidity analog devices are connected to a *APEX*-enabled MSP430 MCU running at 8MHz and synthesized using a Basys3 Artix-7 FPGA board. As shown in Figure 5.5, MCU GPIO ports are connected to the temperature/humidity sensor and to the buzzer. *APEX* is used to prove execution of the fire sensor software. This software is shown in Figure 5.8a in Appendix 5.11. It consists of two main functions: `ReadSensor` and `SoundAlarm`. Proofs of execution are requested by the safety authority via XRequest to issue

[1]https://www.zigbee.org/

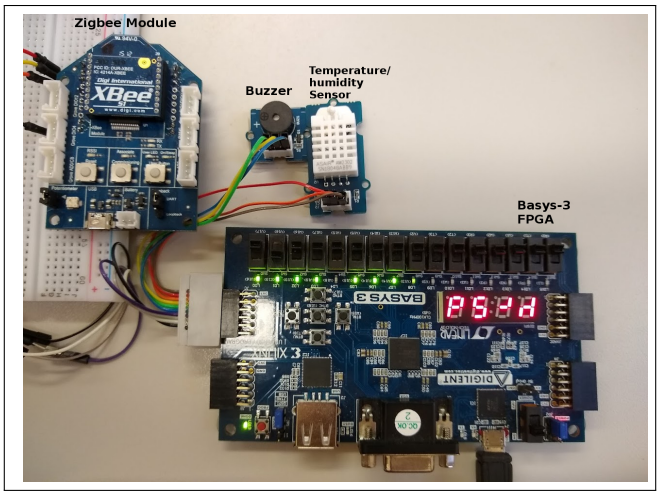


Figure 5.5: Hardware setup for a fire sensor using *APEX*

commands to execute these functions. `ReadSensor` reads and processes the value generated by temperature/humidity analog device memory-mapped GPIO, and copies this value to *OR*. The `SoundAlarm` function turns the buzzer on for 2 seconds, i.e., it writes "1" to the memory address mapped to the buzzer, busy-waits for 2 seconds, and then writes "0" to the same memory location. This implementation was taken from an open-source repository [2] and ported to an *APEX*-enabled MCU. The porting effort was minimal: it involved around 30 additional lines of `C` code, mainly for re-implementing sub-functions originally implemented as shared APIs, e.g., `digitalRead/Write`. Finally, we transformed ported code to be compatible with *APEX*. Details can be found in Appendix 5.11.

5.7 Limitations & Future Directions

In the following we discuss some limitations of *APEX* and directions for future work.

Shared libraries. In order to produce a valid proof, $\mathcal{V}$rf must ensure that the execution of $\mathcal{S}$ does not depend on external code located outside the executable range ER (e.g., shared libraries). A call to such code would violate LTL 5.5, resulting in $EXEC = 0$ during the

[2] https://github.com/Seeed-Studio/LaunchPad_Kit

attestation computation. To support this type of executable one can transform it into a self-contained executable by statically linking all dependencies during the compilation time.

Self-modifying code (SMC). SMC is a type of executable that alters itself while executing. Clearly, this executable type violates LTL 5.4 that requires code in ER to remain unchanged during ER execution. It is unclear how $APEX$ can be adapted to support SMC; however, we are unaware of any realistic useful use-case of SMC in our targeted platforms.

Atomic Execution & Interrupts. The notion of successful execution, defined in Section 5.4.1, prohibits interruptions during ER execution. This limitation can be problematic especially on systems with strict real-time constraints. In this case, the `PoX` executable might be interrupted by a higher priority task and, in order to provide a valid proof of execution, execution of $\mathcal{S}$ must start over. On the other hand, simply resuming $\mathcal{S}$ execution after an interrupt may result in attacks where malware modifies intermediate execution results, in data memory, consequently influencing this execution's output. One way to remedy this issue is to allow interrupts as long as all interrupt handlers are: (1) immutable and uninterruptible from the start of execution until the end of attestation; and (2) included in the attested memory range during the attestation process. $\mathcal{V}\mathsf{rf}$ could then use the `PoX` result $\mathcal{H}$ to determine whether an interrupt that may have happened during the execution is malicious. This idea needs to be examined carefully, because even the accurate definition of `PoX` correctness and security in this case becomes challenging. The practicality and formal security analysis of such an approach also remain an open problem.

Future Directions. There is a number of interesting future directions related to `PoX`. Developing formally verified `PoX` architectures for high-end devices is an interesting challenge. While architectures based on Flicker [83] and SGX [64] can provide `PoX` on high-end devices, the trusted components in these architectures (i.e., TPM and architectural support) are not yet verified. It would also be interesting to investigate whether $APEX$ can be designed and implemented as a standalone device (e.g., a tiny verified TPM-alike device) that can

be plugged into legacy low-end devices. Feasibility and cost-effectiveness of this approach require further investigation; because hybrid-architectures (such as SMART, *VRASED*, and *APEX*) monitor internal MCU signals (e.g., PC, or DMA) that are not exposed to external devices via communication/IO channels. It would also be interesting to see what kinds of trusted applications can be bootstrapped and built on top of a `PoX` service for low-end devices. Finally, in the near-future, we plan to look into techniques that can automatically transform legacy code into `PoX`-compatible software (see Appendix 5.11) and to investigate how to enable stateful `PoX`, where one large `PoX` code could be broken down into multiple smaller pieces of atomic code and secure interruptions are allowed in between the execution of two pieces.

5.8 Conclusion

This chapter introduced *APEX*, the first formally verified `PoX` architecture targeting low-end embedded devices. It allows a remote untrusted prover to generate unforgeable proofs of remote software execution. We envision *APEX* use in many IoT application domains, such as authenticated sensing and actuation. *APEX* is prototyped on a real embedded system platform, MSP430, synthesized on an FPGA, and the verified implementation is publicly available. Our evaluation shows that *APEX* has low overhead for both hardware footprint and time for generating proofs of execution.

APPENDIX

5.9 Appendix: Sub-Module Verification

APEX is designed as a set of seven sub-modules. We now describe *APEX* verified implementation, by focusing on two of these sub-modules and their corresponding properties. The Verilog implementation of omitted sub-modules is available in [2]. Each sub-module enforces a sub-set of the LTL specifications in Definition 14. As discussed in Section 5.5, sub-modules are designed as FSMs. In particular, we implement them as Mealy FSMs, i.e, their output changes as a function of both the current state and current input values. Each FSM takes as input a subset of the signals shown in Figure 5.2 and produces only one output – $EXEC$ – indicating violations of `PoX` properties.

To simplify the presentation, we do not explicitly represent the value of $EXEC$ for each state transition. Instead, we define the following implicit representation:

1. $EXEC$ is 0 whenever an FSM transitions to $NotExec$ state.
2. $EXEC$ remains 0 until a transition leaving $NotExec$ state is triggered.
3. $EXEC$ is 1 in all other states.
4. **Sub-modules composition:** Since all `PoX` properties must simultaneously hold, the value of $EXEC$ produced by *APEX* is the conjunction (logical AND) of all sub-modules' individual $EXEC$ flags.

Figure 5.6 represents a verified model enforcing LTLs 5.5-5.7, corresponding to the high-level property **EP2- Ephemeral Atomicity**. The FSM consists of five states. $notER$ and $midER$ represent states when PC is: (1) outside ER, and (2) within ER respectively, excluding the first (ER_{min}) and last (ER_{max}) instructions. Meanwhile, $fstER$ and $lstER$ correspond to states when PC points to the first and last instructions, respectively. The

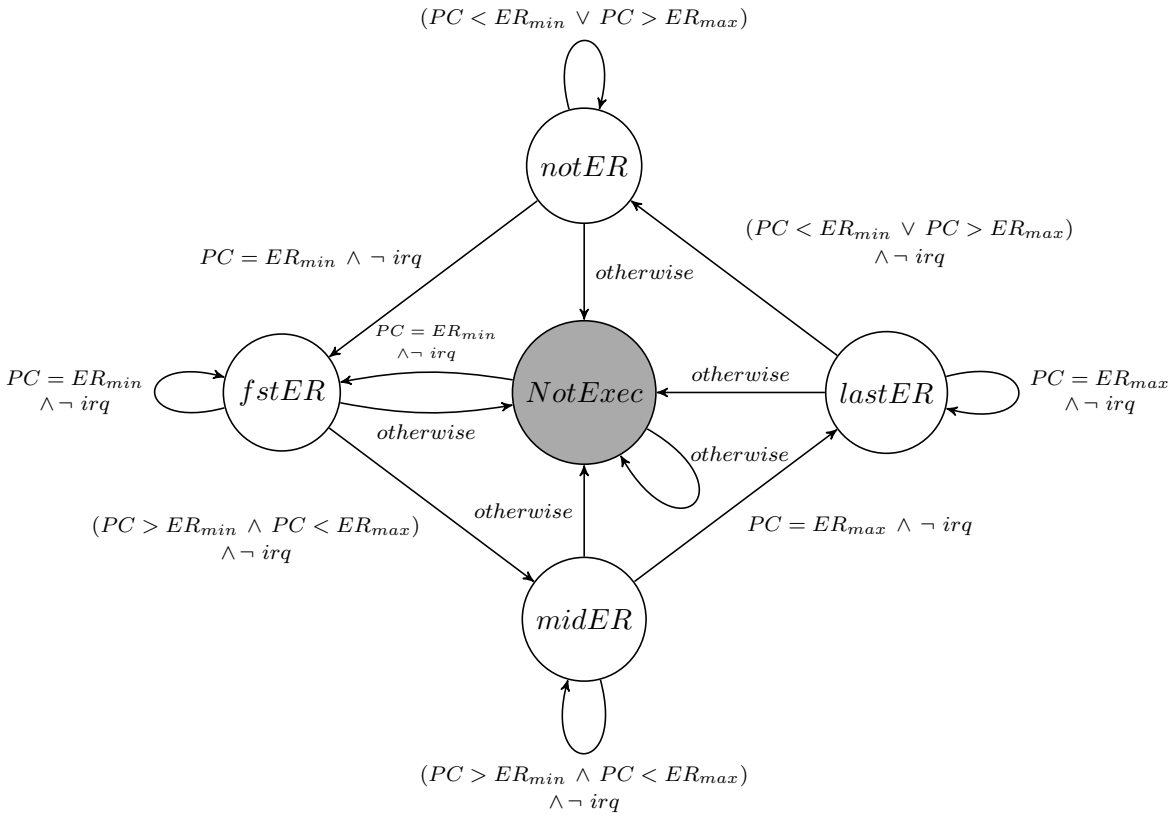


Figure 5.6: Verified FSM for LTLs 5.5-5.7, a.k.a., EP2- Ephemeral Atomicity.

only possible path from $notER$ to $midER$ is through $fstER$. Similarly, the only path from $midER$ to $notER$ is through $lstER$. A transition to the $NotExec$ state is triggered whenever: (1) any sequence of values for PC do not follow the aforementioned conditions, or (2) irq is logical 1 while PC is inside ER. Lastly, the only way to transition out of the $NotExec$ state is to restart ER execution.

Figure 5.7 shows the FSM verified to comply with LTL 5.11 (**MP3- Challenge Temporal Consistency**). The FSM has two states: *Run* and *NotExec*. The FSM transitions to the $NotExec$ state and outputs $EXEC = 0$ whenever a violation happens, i.e., whenever $METADATA$ is modified in software. It transitions back to *Run* when ER execution is restarted without such violation.

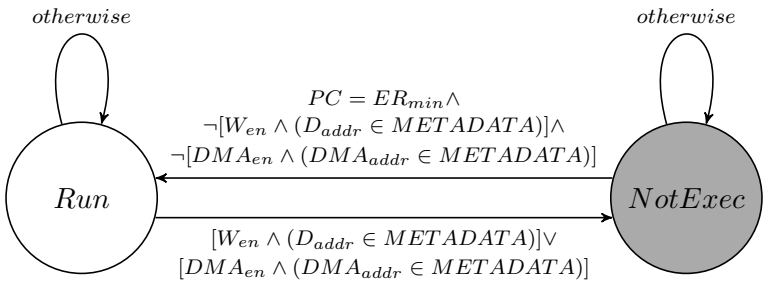


Figure 5.7: Verified FSM for LTL 5.11, a.k.a., MP3- Challenge Temporal Consistency.

5.10 Appendix: Proofs of Implementation Correctness & Security

In this section we discuss the computer proof for *APEX* implementation correctness (Theorem 6) and the reduction, showing that *APEX* is a secure PoX architecture as long as *VRASED* is a secure RA architecture (Theorem 7). A formal LTL computer proof for Theorem 6 is available at [2]. We here discuss the intuition behind such proof. Theorem 6 states that LTLs 5.4 – 5.13, when considered in conjunction with the machine model in Definition 12, imply *APEX* implementation correctness.

Theorem 6. *Definition 12* $\wedge$ *LTLs 5.4 –5.13* $\rightarrow$ *Definition 13.*

Recall that Definition 13 states that, in order to have $EXEC = 1$ during the computation of XProve, at least once **before such event** $(EXEC = 1)$ the following must have happened:

1. The system reached state S_0 in which the software stored in ER started executing from its first instruction ($PC = ER_{min}$).
2. The system eventually reached a state S_1 when ER finished executing ($PC = ER_{max}$). In the interval between S_0 and S_1 PC remained executing instructions within ER, and there were no interrupts, no resets, and DMA remained inactive.
3. The system eventually reached a state S_2 when XProve started executing ($PC =$

CR_{min}). In the interval between S_0 and S_2 the memory regions of $METADATA$ and ER were not modified.

4. In the interval between S_0 and S_2 the OR memory region was only modified by ER software execution ($PC \in ER \vee \neg$ Modify_Mem(OR)).

The first two properties to be noted are LTL 5.13 and LTL 5.12. LTL 5.13 establishes the default state of $EXEC$ is 0. LTL 5.12 enforces that the only possible way to change $EXEC$ from 0 to 1 is by having $PC = ER_{min}$. In other words, $EXEC$ is 1 during the computation of XProve only if, at some point before that, the code stored in ER started to execute (state S_0).

To see why state S_1 (when ER execution finishes, i.e., $PC = ER_{max}$) is reached with ER executing atomically until then, we look at LTLs 5.5, 5.6, 5.7, and 5.10. LTLs 5.5, 5.6 and 5.7 enforce that PC will stay inside ER until S_1 or otherwise $EXEC$ will be set to 0. On the other hand, it is impossible to execute instructions of XProve ($PC \in CR$) without leaving ER, because LTL 5.10 guarantees that ER and CR do not overlap, or $EXEC = 0$.

So far we have argued that to have a token $\mathcal{H}$ that reflects $EXEC = 1$ the code contained in ER must have executed successfully. What remains to be shown is: producing this token implies the code in ER and $METADATA$ are not modified in the interval between S_0 and S_2 and only ER execution can modify OR in the same time interval.

Clearly, the contents of ER can not be modified after S_0 because Modify_Mem(ER) directly implies that LTL 5.4 will set $EXEC = 0$. The same reasoning is applicable for modifications to $METADATA$ region with respect to LTL 5.11. The same argument applies to modifying OR, with the only exception that OR modifications are allowed only by the CPU and when $PC \in ER$ (LTL 5.8). This means that OR can only be modified by the execution of ER. In addition, LTL 5.8 also ensures that DMA is disabled during the execution of ER to prevent unauthorized modification of intermediate results in data memory. Therefore, the timeline

presented in Figure 5.3 is strictly implied by *APEX* implementation. This concludes the reasoning behind Theorem 6.

Theorem 7. *APEX is secure according to Definition 10 as long as VRASED is a secure* RA *architecture according to Definition 15.*

Definition 15. *VRASED Security Game [40]*
15.1 RA ***Security Game (*RA*-game):***
Notation:
- l is the security parameter and $|\mathcal{K}| = |\mathcal{C}\mathsf{hal}| = |MR| = l$
- $AR(t)$ denotes the content of AR at time t
RA-*game:*

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

15.2 RA ***Security Definition:***
An RA *scheme is considered secure if for all PPT adversaries* $\mathcal{A}\mathsf{dv}$*, there exists a negligible function* negl *such that:*

$$Pr[\mathcal{A}\mathsf{dv}, \mathsf{RA}\text{-}game] \leq \mathsf{negl}(l)$$

Proof. (Theorem 7) Assume that $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ is an adversary capable of winning the security game in Definition 10 against *APEX* with more than negligible probability. We show that, if such $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ exists, then it can be used to construct (in a polynomial number of steps) $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ that wins *VRASED* security game (Definition 15) with more than negligible probability. Therefore, by contradiction, nonexistence of $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ (i.e., *VRASED* security) implies nonexistence of $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ (*APEX* security).

First we recall that, to win *APEX* security game, $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ must provide ($\mathcal{H}_{\mathcal{A}\mathsf{dv}}$, $\mathcal{O}_{\mathcal{A}\mathsf{dv}}$), such that $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{O}_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$. To comply with conditions 3.a and 3.b in Definition 10, this must be done in either of the following two ways:

`Case1` $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ does not execute $\mathcal{S}$ in the time window between t_{req} and t_{verif} (i.e., $\neg\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$).

`Case2` $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ calls $\mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}, t_{req} \rightarrow t_{verif})$ but modifies its output $\mathcal{O}$ in between the time when the execution of $\mathcal{S}$ completes and the time when XProve is called.

```c
#define  P4IN          (*(volatile unsigned char *) 0x001C)
#define  P4OUT         (*(volatile unsigned char *) 0x001D)
#define  P4DIR         (*(volatile unsigned char *) 0x001E)
#define  P4SEL         (*(volatile unsigned char *) 0x001F)
#define  BIT4          (0x0010)
#define MAXTIMINGS     85
#define OR             0xEEE0 // OR is in AR
#define HIGH           0x1
#define LOW            0x0
#define INPUT          0x0
#define OUTPUT         0x1
__attribute__ ((section (".exec.entry"), naked)) void ReadSensorEntry() {
    // ERmin
    ReadSensor();
    __asm__ volatile( "br #__exec_leave" "\n\t");
}
__attribute__ ((section (".exec.body"))) int digitalRead() {
    if(P3IN & BIT4) return HIGH;
    else return LOW;
}
__attribute__ ((section (".exec.body"))) void digitalWrite(uint8_t val) {
    if (val == LOW)
        P3OUT &= ~BIT4;
    else
        P3OUT |= BIT4;
}
__attribute__ ((section (".exec.body"))) void pinMode(uint8_t val) {
    if (val == INPUT)
        P3DIR &= ~BIT4;
    else if (val == OUTPUT)
        P3DIR |= BIT4;
}

__attribute__ ((section (".exec.body"))) void ReadSensor() {
    // Tell the sensor that we are about to read
    digitalWrite(HIGH);
    delayMS(250);
    pinMode(OUTPUT);
    digitalWrite(LOW);
    delayMS(20);
    digitalWrite(HIGH);
    delayMicroseconds(40);
    pinMode(INPUT);
    uint8_t laststate = HIGH, counter = 0, j = 0, i;
    uint8_t data[5] = {0};
    // Read the sensor's value
    for ( i=0; i< MAXTIMINGS; i++) {
        counter = 0;
        while (digitalRead() == laststate) {
          counter++;
          if (counter == 255) {
            break;
          }
        }
        laststate = digitalRead();
        if (counter == 255) break;
        if ((i >= 4) && (i%2 == 0)) {
          data[j/8] <<= 1;
          if (counter > 100) {
            data[j/8] |= 1;
            avg += counter;
            k++;
          }
          j++;
        }
    }
    // Copy the reading to OR
    memcpy(OR, data, 5);
}

__attribute__ ((section (".exec.exit"), naked)) void ReadSensorExit() {
    __asm__ volatile("ret" "\n\t");
    // ERmax
}
```

(a) Fire Sensor code written in C

```
1   ...
2   SECTIONS
3   {
4     ...
5     .text :
6     {
7        ...
8       *(.exec.entry)
9       . = ALIGN(2);
10      *(.exec.body)
11      . = ALIGN(2);
12      PROVIDE (__exec_leave = .);
13      *(.exec.exit)
14    } > REGION_TEXT
15    ...
16  }
17  ...
```

(b) Linker script

Figure 5.8: Code snippets for (a) fire sensor described in Section 5.6.3 (b) linker script

However, according to the specification of *APEX* XVerify algorithm (see Definition 11), a token $\mathcal{H}_{\mathcal{A}\mathsf{dv}}$ will only be accepted if it reflects an input value with $EXEC = 1$, as expected by $\mathcal{V}\mathsf{rf}$. In *APEX* implementation, $\mathcal{O}$ is stored in region OR and $\mathcal{S}$ in region ER. Moreover, given Theorem 6, we know that having $EXEC = 1$ during XProve implies three conditions have been fulfilled:

`Cond1` The code in ER executed successfully.

`Cond2` The code in ER and $METADATA$ were not modified after starting ER execution and before calling XProve.

`Cond3` Outputs in OR were not modified after completing ER execution and before calling XProve.

The third condition rules out the possibility of `Case2` since that case assumes $\mathcal{A}\mathsf{dv}$ can modify $\mathcal{O}$, resided in OR, after ER execution and $EXEC$ stays logical 1 during XProve. We further break down `Case1` into three sub-cases:

`Case1.1` $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ does not follow Cond1-Cond3. The only way for $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ to produce ($\mathcal{H}_{\mathcal{A}\mathsf{dv}}$, $\mathcal{O}_{\mathcal{A}\mathsf{dv}}$) in this case is **not** to call XProve and directly guess $\mathcal{H}$.

`Case1.2` $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ follows Cond1-Cond3 but does not execute $\mathcal{S}$ between t_{req} and t_{verif}. Instead, it produces ($\mathcal{H}_{\mathcal{A}\mathsf{dv}}$, $\mathcal{O}_{\mathcal{A}\mathsf{dv}}$) by calling:

$$\mathcal{O}_{\mathcal{A}\mathsf{dv}} \equiv \mathsf{XAtomicExec}^{\mathcal{P}\mathsf{rv}}(ER_{\mathcal{A}\mathsf{dv}}, t_{req} \rightarrow t_{verif}) \tag{5.14}$$

where $ER_{\mathcal{A}\mathsf{dv}}$ is a memory region different from the one specified by $\mathcal{V}\mathsf{rf}$ on $\mathsf{XRequest}$ ($\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ can do this by modifying $METADATA$ to different values of ER_{min} and ER_{max} before calling $\mathsf{XAtomicExec}$).

`Case1.3` Similar to Case1.2, with $ER_{\mathcal{A}\mathsf{dv}}$ being the same region specified by $\mathcal{V}\mathsf{rf}$ on $\mathsf{XRequest}$, but instead containing a different executable $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$.

We show that an adversary that succeeds in any of these cases can be used win *VRASED* security game. To see why this is the case, we note that *APEX* XProve function is implemented by using *VRASED* `SW-Att`. `SW-Att` covers memory regions MR (challenge memory) and AR (attested region). Hence, *APEX* instantiates these memory regions as:

1. $MR = \mathcal{C}\mathsf{hal}$;
2. $ER \subset AR$;
3. $OR \subset AR$;
4. $METADATA \subset AR$;

Doing so ensures that all sensitive memory regions used by *APEX* are included among the inputs to *VRASED* attestation. Let $X(t)$ denote the content in memory region X at time t. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ can then be constructed using $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ as follows:

1. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ receives $\mathcal{C}\mathsf{hal}$ from the challenger in step (2) of RA security game of Definition 15.
2. At arbitrary time t, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ has 3 options to write $AR(t) = AR_{\mathcal{A}\mathsf{dv}}$ and call $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$:
   (a) Modify $ER(t) \neq \mathcal{S}$ or $OR(t) \neq \mathcal{O}$ or $METADATA(t) \neq METADATA_{\mathcal{V}\mathsf{rf}}$. It then calls $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ in `Case1.1`.
   (b) Modify ER to be different from the range chosen by $\mathcal{V}\mathsf{rf}$. Therefore, $METADATA(t) \neq METADATA_{\mathcal{V}\mathsf{rf}}$. It then calls $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ in `Case1.2`.
   (c) Modify $ER(t)$ to be different from $\mathcal{S}$. It then calls $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ in `Case1.3`.

   In any of these options, $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ will produce ($\mathcal{H}_{\mathcal{A}\mathsf{dv}}$,$\mathcal{O}_{\mathcal{A}\mathsf{dv}}$), such that $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{O}_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$ with non-negligible probability.
3. $\mathcal{A}\mathsf{dv}_{\mathsf{RA}}$ replies to the challenger with the pair $(M, \mathcal{H}_{\mathcal{A}\mathsf{dv}})$, where M corresponds to the values of

$\mathcal{S}$, $\mathcal{O}$ and $METADATA_{\mathcal{V}\mathsf{rf}}$, matching $\mathcal{H}_{\mathcal{A}\mathsf{dv}}$ and $\mathcal{O}_{\mathcal{A}\mathsf{dv}}$ generated by $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$. By construction $M \neq AR_{\mathcal{A}\mathsf{dv}} = AR(t)$, as required by Definition 15.

4. Challenger will accept $(M, \mathcal{H}_{\mathcal{A}\mathsf{dv}})$ with the same non-negligible probability that $\mathcal{A}\mathsf{dv}_{\mathtt{PoX}}$ has of producing ($\mathcal{H}_{\mathcal{A}\mathsf{dv}}$,$\mathcal{O}_{\mathcal{A}\mathsf{dv}}$) such that $\mathsf{XVerify}(\mathcal{H}_{\mathcal{A}\mathsf{dv}}, \mathcal{O}_{\mathcal{A}\mathsf{dv}}, \mathcal{S}, \mathcal{C}\mathsf{hal}, \cdot) = 1$.

□

5.11 Appendix: Software Transformation

Recall that the notion of successful execution (in Section 5.4.1) requires the executable's entry point to be at the first instruction in ER and the exit point to be at the last instruction in ER. In this section, we discuss how to efficiently transform arbitrary software to conform with this requirement.

Lines 10-17 of Figure 5.8.a show a (partial) implementation of the `ReadSensor` function described in Section 5.6.3. This implementation, when converted to an executable, does not meet *APEX* executable requirement, since the compiler may choose to place one of its sub-functions (instead of `ReadSensor`) as the entry and/or exit points of the executable. One way to fix this issue is to implement all of its sub-functions as inline functions. However, this may be inefficient; in this example, it would duplicate the code of the same sub-functions (e.g., `digitalWrite`) inside the executable.

Instead, we create dedicated functions for entry (Line 1-4) and exit (Line 6-8) points, and assign those functions to separate executable sections: ".exec.entry" for the entry and ".exec.exit" for the exit. Then, we label all sub-functions used by `ReadSensor` as well as `ReadSensor` itself to the same section – ".exec.body" – and modify the MSP430 linker to place ".exec.body" between ".exec.entry" and ".exec.exit" sections. The modified linker script is shown in Figure 5.8.b.

Chapter 6

TAROT: Trigger-based Active Root Of Trust

Abstract

Several tiny Roots-of-Trust (RoTs) – including the ones described in previous chapters – were proposed with the purpose of securing low-end devices. In general, these RoTs operate reactively: they can prove whether a desired action (e.g., software update, or execution of a program) was performed on a specific device. However, they can not guarantee that a desired action **will be performed**, since malware controlling the device can trivially block access to the RoT by ignoring/discarding received commands and other trigger events. This is a major and important problem because it allows malware to effectively "brick" or incapacitate a potentially huge number of (possibly mission-critical) devices. Though recent work made progress in terms of incorporating more active behavior atop existing RoTs, it relies on extensive hardware support –Trusted Execution Environments (TEEs) which are too costly for low-end devices. In this chapter, we set out to systematically design a minimal active RoT for tiny low-end MCU-s. We begin with the following questions: (1) What functions and hardware support are required to guarantee actions in the presence of malware?, (2) How to implement this efficiently?, and (3) What security benefits stem from such an active RoT architecture? We then design, implement, formally verify, and evaluate *TAROT* : Trigger-based Active Root-Of-Trust. We believe that *TAROT* is the first clean-slate design of an active RoT for low-end MCU-s. We show how *TAROT* guarantees that even a fully software-compromised low-end MCU performs a desired action. We demonstrate its practicality by implementing *TAROT* in the context of three types of applications where actions are triggered by: sensing hardware, network events, and timers. We also formally specify and verify *TAROT* functionality and properties. The contributions described in this chapter are currently in submission to the USENIX Security Symposium – 2022 (see pre-print in [7]).

6.1 Introduction

Numerous architectures focused on securing low-end micro-controller units (MCU-s) by designing small and affordable trust anchors [5]. However, most such techniques **operate passively**. They can prove, to a trusted party, that certain property (or action) is satisfied (or was performed) by a remote and potentially compromised low-end MCU. Examples of such services include remote attestation [49, 87, 40, 10, 22, 68], proofs of remote software execution [42], control-flow & data-flow attestation [44, 6, 45, 116, 105, 43], as well as proofs of remote software update, memory erasure, and system-wide reset [41, 9, 15].

Aforementioned approaches are passive in nature. While they can detect integrity violations of remote devices, they cannot guarantee that a given security or safety-critical task will be performed. For example, consider a network comprised of a large number (of several types of) simple IoT devices, e.g., an industrial control system. Upon detecting a large-scale compromise, a trusted remote controller must fix the situation by requiring all compromised devices to reset, erase, or update themselves in order to expunge malware. Even if each device has a passive RoT, malware (which is in full control of the device's software state) can easily intercept, ignore, or discard any requests for the RoT, thus preventing its functionality from being triggered. Therefore, the only way to repair these compromised devices requires direct physical access (i.e, reprogramming by a human) to each device. Beyond the DoS damage caused by the multitude of essentially "bricked" devices, physical reprogramming itself is slow and disruptive, i.e., a logistical nightmare. Also, some devices might be deployed in locations that are difficult to access physically. (For example, under water, encased in walls or foundations, in the air or in space.)

Motivated by the above, recent research [115, 60] constructed trust anchors with a more active behavior. Xu et al. [115] propose the concept of Authenticated Watch-Dog Timers (WDT), which can enforce periodic execution of a secure component (an RoT task), unless

explicit authorization (which can itself include a set of tasks) is received from a trusted controller. In [60] this concept is realized with the reliance on an existing passive RoT (ARM TrustZone), as opposed to a dedicated co-processor as in the original approach from [115]. Both techniques are time-based, periodically and actively triggering RoT invocation, despite potential compromise of the host device. (We discuss them in more detail in Section 6.4.4.)

In this chapter, we take the next step and design a more general active RoT, called *TAROT*: Trigger-based Active Root-Of-Trust. Our goal is guaranteed execution of trusted and safety-critical tasks based on arbitrary events captured by hardware peripherals (e.g., timers, GPIO ports, and network interfaces) of an MCU the software state of which may be currently compromised. In principle, any hardware event that can be configured to cause an interruption on the unmodified MCU can be used to trigger guaranteed execution of trusted software in *TAROT*. In that vein, our work can be viewed as a generalization of concepts proposed in [115, 60], enabling arbitrary events to trigger guaranteed execution of trusted functionalities. In comparison, prior work has the advantage of relying on existent hardware. On the other hand, our clean-slate approach, based on minimal hardware design, enables new applications and is applicable to low-end resource-constrained MCU-s.

At a high level, *TAROT* is based on two main notions: "**Guaranteed Triggering**" and "**Re-Triggering on Failure**". The term trigger is used to refer to an event that causes *TAROT* to take over the execution in the MCU. The "guaranteed triggering" property ensures that a particular event of interest always triggers execution of *TAROT*. Whereas, "re-triggering on failure" assures that, if RoT execution is illegally interrupted for any reason (e.g., attempts to violate execution's integrity, power faults, or resets), the MCU resets and the RoT is guaranteed to be the first to execute after subsequent re-initialization. Figure 6.1 illustrates this workflow.

We use *TAROT* to address three realistic and compelling use-cases:

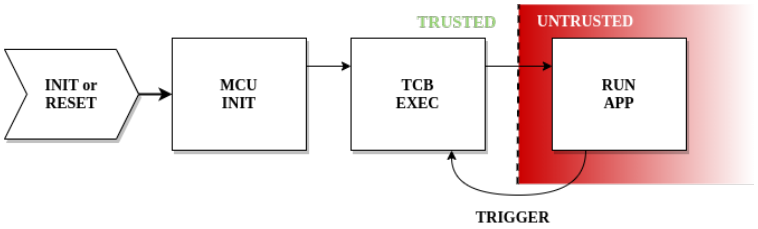


Figure 6.1: *TAROT* Software Execution Flow

- GPIO-TCB: A safety-critical sensor/actuator hybrid, guaranteed to sound an alarm if the sensed quantity (e.g., temperature, CO_2 level, etc) exceeds a certain threshold. This use-case exemplifies hardware-based triggering.
- TimerTCB: A real-time system where a predefined safety-critical task is guaranteed to execute periodically. This use-case exemplifies timer-based triggering, which is also attainable by [115, 60].
- NetTCB: a trusted component that is always guaranteed to process commands received over the network, thus preventing malware in the MCU from intercepting and/or discarding commands destined for the RoT. This use-case exemplifies network-based triggering.

In all three cases, the guarantees hold despite potential full compromise of the MCU software state, as long as the RoT itself is trusted.

In addition to designing and instantiating *TAROT* with three use-cases, we formally specify *TAROT* goals and requirements using Linear Temporal Logic (LTL). These formal specifications offer precise definitions for the security offered by *TAROT* and its corresponding assumptions expected from the underlying machine model. This can serve as a unambiguous reference for future implementations and for other derived services. Finally, we use formal verification to prove that the implementation of *TAROT* hardware modules adheres to a set of sub-properties (also specified in LTL) that – when composed with the MCU machine model – are sufficient to achieve *TAROT* end-to-end goals.

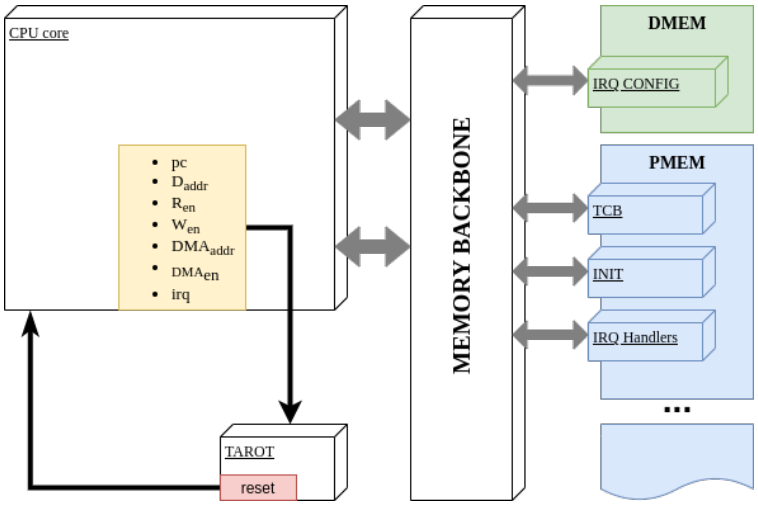


Figure 6.2: *TAROT* in the MCU architecture

We implement and evaluate *TAROT* and make its verified implementation along with respective computer proofs/formal verification publicly available in [3].

6.2 *TAROT* Overview

The goal of *TAROT* is to guarantee eventual execution of a pre-defined functionality $\mathcal{F}$ implemented as a trusted software executable. We refer to this executable as *TAROT* trusted computing base (TCB). *TAROT* is agnostic to the particular functionality implemented by $\mathcal{F}$, which allows guaranteed execution of arbitrary tasks, to be determined based on the application domain; see Section 6.4 for examples.

A trigger refers to a particular event that can be configured to cause the TCB to execute. Examples of possible triggers include hardware events from:

- *External (usually analog) inputs, e.g., detection of a button press, motion, sound or certain temperature/CO_2 threshold.*
- *Expiring timers, i.e., a periodic* trigger.
- *Arrival of a packet from the network, e.g., carrying a request to collect sensed data, perform*

sensing/actuation, or initiate a security task, such as code update or remote attestation.

If configured correctly, these events cause interrupts, which are used by *TAROT* to guarantee execution of $\mathcal{F}$. Since trigger and the TCB implementation are configurable, we assume that these initial configurations are securely provisioned before initial deployment. The trigger configuration includes the types of interruptions and respective settings e.g., GPIO port, type of event, its time granularity, etc. At runtime, *TAROT* protects the initial configuration from illegal modifications, i.e., ensures correct trigger behavior. This protection includes preserving interrupt configuration registers, interrupt handlers (a.k.a., interrupt service routines – ISRs), and interrupt vectors. This way *TAROT* guarantees that trigger always results in an invocation of the TCB.

However, guaranteed invocation of the TCB upon occurrence of a trigger is not sufficient to claim that $\mathcal{F}$ is properly performed, since the TCB code (and execution thereof) could itself be tampered with. To this end, *TAROT* provides runtime protections that prevent any unprivileged/untrusted program from modifying the TCB code, i.e., the program memory region reserved for storing that code. (Recall that instructions execute in place, from program memory). *TAROT* also monitors the execution of the TCB code to ensure:

1. **Atomicity:** Execution is atomic (i.e., uninterrupted), from the TCB first instruction (legal entry), to its last instruction (legal exit);
2. **Non-malleability:** During execution, $DMEM$ cannot be modified, other than by the TCB code itself, e.g., no modifications by other software or DMA controllers.

These properties ensure that any potential malware (i.e., compromised software outside the TCB or compromised DMA controllers) cannot tamper with the TCB execution.

TAROT monitors the TCB execution and, if a violation of *any* property (not just atomicity and non-malleability of the TCB, but also illegal trigger misconfiguration) occurs, it causes an immediate MCU reset to a default trusted state where the TCB code is the first

component to execute. Therefore, any attempt to interfere with the TCB functionality or execution only causes the TCB to recover and re-execute, this time with the guarantee that unprivileged/untrusted applications cannot interfere.

Both trigger configurations and the TCB implementation are updatable at run-time, as long as the updates are performed from within the TCB itself. While this feature is not strictly required for security, we believe it provides flexibility/updatability, while ensuring that untrusted software is still unable to modify *TAROT* trusted components and configuration thereof. In Section 6.3.6, we also discuss how *TAROT* can enforce confidentiality to the TCB task, which is applicable to cases where $\mathcal{F}$ implements privacy-sensitive tasks.

Each sub-property in *TAROT* is implemented as a separate sub-module. These sub-modules are then composed and shown secure (when applied to the MCU machine model) using a combination of model-checking-based formal verification and an LTL computer-checked proof. *TAROT* modular design enables verifiability and minimality, resulting in low hardware overhead and significantly higher confidence about the security provided by its design and implementation.

As shown in Figure 6.2, *TAROT* is implemented as a hardware component that monitors a set of CPU signals to detect violations of required security properties. It does not interfere with the CPU core implementation, e.g., by modifying its behavior or instruction set. In subsequent sections we describe these properties in detail and discuss their implementation and verification. Finally, we use a commodity FPGA to implement *TAROT* atop MSP430 and report on its overhead.

PC	Current Program Counter value
R_{en}	Signal that indicates if the MCU is reading from memory (1-bit)
W_{en}	Signal that indicates if the MCU is writing to memory (1-bit)
D_{addr}	Address for an MCU memory access (read or write)
DMA_{en}	Signal that indicates if DMA is currently enabled (1-bit)
DMA_{addr}	Memory address being accessed by DMA, if any
gie	Global Interrupt Enable: signal that indicates whether or not interrupts are globally enabled (1-bit).
irq	Signal that indicates if an interrupt is happening
$DMEM$	Region corresponding to the entire data memory of the MCU: $DMEM = [DMEM_{min}, DMEM_{max}]$.
$PMEM$	Region corresponding to the entire program memory of the MCU: $PMEM = [PMEM_{min}, PMEM_{max}]$.
TCB	Memory region reserved for the TCB's executable implementing $\mathcal{F}$. $TCB = [TCB_{min}, TCB_{max}]$. $TCB \in PMEM$.
$INIT$	Memory region containing the MCU's default initialization code. $INIT = [INIT_{min}, INIT_{max}]$. $INIT \in PMEM$.
$reset$	A 1-bit signal that reboots/resets the MCU when set to logical 1

Table 6.1: Summary of *TAROT*-relevant notation

6.3 *TAROT* in Detail

6.3.1 Notation, Machine Model, & Assumptions

This section discusses our machine and adversary models. For quick-reference, Table 6.1 summarizes the notation used in the rest of the chapter.

CPU Hardware Signals

TAROT neither modifies nor verifies the underlying CPU core/instruction set. It is assumed that the underlying CPU adheres to its specification and *TAROT* is implemented as a standalone hardware module that runs in parallel with the CPU, and enforcing necessary guarantees in hardware. The following CPU signals are relevant to *TAROT*:

H1 – ***Program Counter (PC):*** PC always contains the address of the instruction being executed in the current CPU cycle.

H2 – ***Memory Address:*** Whenever memory is read or written by the CPU, the data-address signal (D_{addr}) contains the address of the corresponding memory location. For a read access, a data read-enable bit (R_{en}) must be set, while, for a write access, a data write-enable bit (W_{en}) must be set.

H3 – ***DMA:*** Whenever a DMA controller attempts to access the main system memory, a DMA-address signal (DMA_{addr}) reflects the address of the memory location being accessed and a DMA-enable bit (DMA_{en}) must be set. DMA can not access memory when DMA_{en} is off (logical zero).

H4 – ***MCU Reset:*** At the end of a successful reset routine, all registers (including PC) are set to zero before resuming normal software execution flow. Resets are handled by the MCU in hardware. Thus, the reset handling routine can not be modified. Once execution re-starts, PC is set to point to the first instruction in the boot section of program memory, referred to as $INIT$ (see M2 below). When a reset happens, the corresponding *reset* signal is set. The same signal is also set when the MCU initializes for the first time. An MCU *reset* also resets its DMA controller, and any prior configuration thereof. (DMA) behavior is configured by user software at runtime. By default (i.e., after a reset) DMA is inactive.

H5 – ***Interrupts:*** Whenever an interrupt occurs, the corresponding *irq* signal is set. Interrupts may be globally enabled or disabled in software. The 1-bit signal *gie* always reflects whether or not they are currently enabled. The default *gie* state (i.e., at boot or after a reset) is disabled (logical zero).

Memory: Layout & Initial Configuration

As far as MCU initial memory layout and its initial software configuration (set at, or prior to, its deployment), the following are relevant to $TAROT$:

M1 – ***PMEM:*** Corresponds to the entire $PMEM$ address space. Instructions are executed in place. Hence, at runtime, PC points to the $PMEM$ address storing the instruction being executed.

M2 – ***INIT:*** Section of $PMEM$ containing the MCU boot segment, i.e., the first software to be executed whenever the MCU boots or after a *reset*. We assume $INIT$ code is finite.

M3 – ***TCB:*** Section of $PMEM$ reserved for *TAROT* trusted code $\mathcal{F}$. TCB is located immediately after $INIT$; it is the first software to execute following successful completion of $INIT$.

M4 – ***IRQ-Table and Handlers:*** IRQ-Table is located in $PMEM$ and contains pointers to the addresses of *interrupt handlers*. When an interrupt occurs, MCU hardware forces a "jump" to the corresponding handler routine (a.k.a. ISR). The address of this routine is specified by the IRQ-Table fixed index corresponding to that particular interrupt. Handler routines are code segments (functions) also stored in $PMEM$.

M5 – IRQ_{cfg}***:*** Set of registers in $DMEM$ used to configure specific behavior of individual interrupts at runtime, e.g., deadline of a timer-based interrupt, or type of event on a hardware-based interrupt.

Note that the initial memory configuration can be changed at run-time (e.g., by malware that infects the device), unless it is explicitly protected by *TAROT* verified hardware modules.

Initial Trigger Configuration

T1 – trigger **configuration:** *TAROT* trigger is configured, at MCU (pre)deployment-time, by setting the corresponding entry in IRQ-Table and respective handler to jump to the first instruction in the TCB (TCB_{min}) and by configuring the registers in IRQ_{cfg} with desired interrupt parameters, reflecting the desired trigger behavior; see Section 6.4 for examples.

Thus, a trigger event causes the TCB code to execute, as long as the initial configuration is maintained.

T2 – trigger priority: *TAROT* trigger-s leverage MCU interrupts. We assume that interrupts used as trigger-s are configured with higher priority over application interrupts. As such, in the case where a *TAROT* safety critical trigger and a regular application interrupt occur at the same clock cycle, *TAROT* trigger is handled by the CPU first. Different MCUs have different ways to set interrupt priorities. In our MSP430-based implementation, this is achieved by placing *TAROT* trigger-s first in the hardware layout of the IRQ-Table, according to MSP430 specification.

T3 – trigger clock source (time-based trigger-s): We assume that the clock sources used to implement the timers used by *TAROT* timer-based trigger-s cannot be disabled by software. Many MCUs offer multiple clocks. Some of them can be disabled by going into low-power modes. Other clock sources are always enabled. We assume that any timer-based trigger is configured (via initial setup of IRQ_{cfg}) to use the latter option.

Our initial configuration is not much different from a regular interrupt configuration in a typical embedded system program. It must correctly point to *TAROT* TCB entry point, just as regular interrupts must correctly point to their respective handler entry points. For example, to initially configure a timer-based trigger, the address in IRQ-Table corresponding to the respective hardware timer is set to point to TCB_{min} and the correspondent registers in IRQ_{cfg} are set to define the desired interrupt period.

Adversary Model

In comparison with previous chapters, TAROT adversary model makes one additional assumption: that $\mathcal{A}$dv can not access $\mathcal{P}$rv physically (for instance, to re-program $\mathcal{P}$rv's software physically, via wired interface). TAROT complete adversary model is described below.

We consider an adversary $\mathcal{A}\mathsf{dv}$ that controls $\mathcal{P}\mathsf{rv}$'s entire software state, including code, and data. $\mathcal{A}\mathsf{dv}$ can read/write from/to any memory that is not explicitly protected by hardware-enforced access control rules. $\mathcal{A}\mathsf{dv}$ might also have full control over all Direct Memory Access (DMA) controllers of $\mathcal{P}\mathsf{rv}$. Recall that DMA allows a hardware controller to directly access main memory ($PMEM$ or $DMEM$) without going through the CPU.

Physical Attacks: physical and hardware-focused attacks are out of the scope of *TAROT*. Specifically, we assume that $\mathcal{A}\mathsf{dv}$ can not modify, induce hardware faults, or interfere with *TAROT* via physical presence attacks and/or side-channels. Protection against such attacks is an orthogonal issue, which can be addressed via physical security techniques [95]. *Different from previous chapters*, we here assume that $\mathcal{A}\mathsf{dv}$ can not access and re-program or re-configure $\mathcal{P}\mathsf{rv}$ physically (e.g., via wired re-writing of program memory, which requires physical presence).

Network DoS Attacks: we also consider out-of-scope all network DoS attacks whereby $\mathcal{A}\mathsf{dv}$ drops traffic to/from $\mathcal{P}\mathsf{rv}$, or floods $\mathcal{P}\mathsf{rv}$ with traffic, or simply jams communication. Note that this assumption is relevant only to network-triggered events, exemplified by the NetTCB instantiation of *TAROT*, described in Section 6.4.3.

Correctness of the TCB Executable: we recall that the main goal of *TAROT* is guaranteed execution of $\mathcal{F}$, as specified by the application developer and loaded onto *TAROT* TCB at provisioning (prior to deployment). Similar to existing RoTs (e.g., TEE-s in higher-end CPUs) *TAROT* does **not** check correctness and absence of implementation bugs in $\mathcal{F}$ implementation. In many applications, $\mathcal{F}$ code is minimal; see examples in Section 6.4. Moreover, correctness of $\mathcal{F}$ need **not** be assured by *TAROT* architecture, within $\mathcal{P}\mathsf{rv}$, at run-time. Since embedded applications are originally developed on more powerful devices (e.g., general-purpose computers), various vulnerability detection methods, e.g., fuzzing [33], static analysis [38], or formal verification, can be employed to avoid or detect implementation bugs in $\mathcal{F}$. All that can be performed off-line before loading $\mathcal{F}$ onto *TAROT* TCB and the entire

issue is orthogonal to *TAROT*.

Machine Model (Formally)

Definition 16. *Machine Model:*
Memory Modifications:

$$\mathbf{G}:\{\mathsf{modMem}(X) \rightarrow (W_{en} \wedge D_{addr} \in X) \vee (DMA_{en} \wedge DMA_{addr} \in X)\} \tag{6.1}$$

Successful Trigger Modification:

$$\mathbf{G}:\{\mathsf{mod}(\mathsf{trigger}_{cfg}) \rightarrow [(\mathsf{modMem}(PMEM) \vee \mathsf{modMem}(IRQ_{cfg})) \wedge \neg reset]\} \tag{6.2}$$

Successful Interrupt Disablement:

$$\boldsymbol{G}\text{:}\{\mathsf{disable}(irq) \rightarrow [\neg reset \wedge gie \wedge \neg\mathbf{X}(gie) \wedge \neg\mathbf{X}(reset)]\} \tag{6.3}$$

Trigger/TCB Initialization (6.4 & 6.5):

$$\boldsymbol{G}\text{:}\{\neg\mathsf{mod}(\mathsf{trigger}_{cfg}) \vee PC \in TCB\} \wedge \boldsymbol{G}\text{:}\{\neg\mathsf{disable}(irq) \vee \mathbf{X}(PC) \in TCB\} \rightarrow \boldsymbol{G}\text{:}\{\mathsf{trigger} \rightarrow F(PC = TCB_{min})\} \tag{6.4}$$

$$\boldsymbol{G}\text{:}\{\neg\mathsf{modMem}(PMEM) \vee PC \in TCB\} \rightarrow \boldsymbol{G}\text{:}\{reset \rightarrow F(PC = TCB_{min})\} \tag{6.5}$$

Figure 6.3: MCU machine model (subset) in LTL.

Based on the high-level properties discussed earlier in this section, we now formalize the subset (relevant to *TAROT*) of the MCU machine model using LTL. Figure 6.3 presents it as a set of LTL statements.

LTL statement (6.1) models the fact that modifications to a given memory address (X) can be done either via the CPU or DMA. Modifications by the CPU imply setting $W_{en} = 1$ and $D_{addr} = X$. If X is a memory region, rather than a single address, we denoted that a modification happened within the particular region by saying that $D_{addr} \in X$, instead. Conversely, DMA modifications to region X require $DMA_{en} = 1$ and $DMA_{addr} \in X$. This models the MCU behaviors stated informally in **H2** and **H3**.

In accordance with **M4** and **M5**, a successful modification to a pre-configured trigger implies changing interrupt tables, interrupt handlers, or interrupt configuration registers (ICR-s). Since, per **M4**, the first two are located in $PMEM$, modifying them means writing to $PMEM$. The ICR is located in a $DMEM$ location denoted IRQ_{cfg}. Therefore, the LTL

statement (6.2) models a successful misconfiguration of trigger as requiring a memory modification either within $PMEM$ or within IRQ_{cfg}, without causing an immediate system-wide reset ($\neg reset$). This is because an immediate reset prevents the modification attempt from taking effect (see **H4**).

LTL (6.3) models that attempts to disable interrupts are reflected by gie CPU signal (per **H5**). In order to successfully disable interrupts, one must switch interrupts from enabled ($gie = 1$) to disabled ($\neg \mathbf{X}(gie)$ – disabled in the following cycle), without causing an MCU $reset$.

Recall that (from **H1**) PC reflects the address of the instruction currently executing. $PC \in TCB$ implies that $TAROT$ TCB is currently executing. LTL 6.4 models **T1**. As long as the initial proper configuration of trigger is never modifiable by untrusted software ($\mathbf{G}$:$\{\neg \mathsf{mod}(\mathsf{trigger}_{cfg}) \vee PC \in TCB\}$) and that untrusted software can never globally disable interrupts ($\mathbf{G}$:$\{\neg \mathsf{disable}(irq) \vee \mathbf{X}(PC) \in TCB\}$), a trigger would always cause the TCB execution ($\mathbf{G}$:$\{\mathsf{trigger} \rightarrow F(PC = TCB_{min})\}$). Recall that we assume that the TCB may update – though not misconfigure – trigger behavior, since the TCB is trusted. Similarly, LTL 6.5 states that, as long as $PMEM$ is never modified by untrusted software, a $reset$ will always trigger the TCB execution (per **H4**, **M2**, and **M3**).

6.3.2 *TAROT* End-To-End Goals Formally

Definition 17. *Guaranteed Trigger:*

$\boldsymbol{G}$:$\{\mathsf{trigger} \rightarrow \mathbf{F}(PC = TCB_{min})\}$

Definition 18. *Re-Trigger on Failure:*

$\boldsymbol{G}$:$\{PC \in TCB \rightarrow [\ (\neg irq \wedge \neg dma_{en} \wedge PC \in TCB) \quad \boldsymbol{W} \quad (PC = TCB_{max} \vee \boldsymbol{F}(PC = TCB_{min})\ ]\}$

Figure 6.4: Formal Specification of $TAROT$ end-to-end goals.

Using the notation from Section 6.3.1, we proceed with the formal specification of $TAROT$

end-goals in LTL. Definition 17 specifies the "guaranteed trigger" property. It states in LTL that, whenever a trigger occurs, a TCB execution/invocation (starting at the legal entry point) will follow.

While Definition 17 guarantees that a particular interrupt of interest (trigger) will cause the TCB execution, it does not guarantee proper execution of the TCB code as a whole. The "re-trigger on failure" property (per Definition 18) stipulates that, whenever the TCB starts execution (i.e., $PC \in TCB$), it must execute without interrupts or DMA interference [1], i.e., $\neg irq \wedge \neg dma_{en} \wedge PC \in TCB$. This condition must hold until:

1. $PC = TCB_{max}$: the legal exit of the TCB is reached, i.e., execution concluded successfully.

2. $F(PC = TCB_{min})$: another TCB execution (from scratch) has been triggered to occur.

In other words, this specification reflects a cyclic requirement: either the security properties of the TCB proper execution are not violated, or the TCB execution will re-start later.

Note that we use the quantifier Weak Until (**W**) instead regular Until (**U**), because, for some embedded applications, the TCB code may execute indefinitely; see Section 6.4.1 for one such example.

6.3.3 *TAROT* Sub-Properties

Based on our machine model and *TAROT* end goals, we now postulate a set of necessary sub-properties for *TAROT*. Next, Section 6.3.4 shows that this minimal set of sub-properties suffices to achieve *TAROT* end-to-end goals with a computer-checked proof. LTL specifications of the sub-properties are presented in Figure 6.5.

[1] Since DMA could tamper with intermediate state/results in $DMEM$.

Definition 19. *LTL Sub-Properties implemented and enforced by TAROT.*

Trusted $PMEM$ ***Updates:***

$$\mathbf{G}: \{[\neg(PC \in TCB) \wedge W_{en} \wedge (D_{addr} \in PMEM)] \vee [DMA_{en} \wedge (DMA_{addr} \in PMEM)] \rightarrow reset\} \tag{6.6}$$

IRQ Configuration Protection:

$$\mathbf{G}: \{[\neg(PC \in TCB) \wedge W_{en} \wedge (D_{addr} \in IRQ_{cfg})] \vee [DMA_{en} \wedge (DMA_{addr} \in IRQ_{cfg})] \rightarrow reset\} \tag{6.7}$$

Interrupt Disablement Protection:

$$\mathbf{G}: \{\neg reset \wedge gie \wedge \neg \boldsymbol{X}(gie) \rightarrow (\boldsymbol{X}(PC) \in TCB) \vee \boldsymbol{X}(reset)\} \tag{6.8}$$

TCB Execution Protection:

$$\mathbf{G}: \{\neg reset \wedge (PC \in TCB) \wedge \neg(\mathbf{X}(PC) \in TCB) \rightarrow PC = TCB_{max} \vee \ \mathbf{X}(reset)\ \} \tag{6.9}$$

$$\mathbf{G}: \{\neg reset \wedge \neg(PC \in TCB) \wedge (\mathbf{X}(PC) \in TCB) \rightarrow \mathbf{X}(PC) = TCB_{min} \vee \ \mathbf{X}(reset)\} \tag{6.10}$$

$$\mathbf{G}: \{(PC \in TCB) \wedge (irq \vee dma_{en}) \rightarrow reset\} \tag{6.11}$$

Figure 6.5: Formal specification of sub-properties verifiably implemented by *TAROT* hardware module.

TAROT enforces that only trusted updates are allowed to $PMEM$. *TAROT* hardware issues a system-wide MCU *reset* upon detecting any attempt to modify $PMEM$ at runtime, unless this modification comes from the execution of the TCB code itself. This property is formalized in LTL (6.6). It prevents any untrusted application software from misconfiguring IRQ-Table and interrupt handlers, as well as from modifying the $INIT$ segment and the TCB code itself, because these sections are located within $PMEM$. As a side benefit, it also prevents attacks that attempt to physically wear off Flash (usually used to implement $PMEM$ in low-end devices) by excessively and repeatedly overwriting it at runtime. Similarly, *TAROT* prevents untrusted components from modifying IRQ_{cfg} – $DMEM$ registers controlling the trigger configuration. This is specified by LTL 6.7.

LTL 6.8 enforces that interrupts can not be globally disabled by untrusted applications. Since, each trigger is based on interrupts, disablement of all interrupts would allow untrusted software to disable the trigger itself, and thus the active behavior of *TAROT*. This requirement is specified by checking the relation between current and next values of *gie*, using the LTL ne**X**t operator. In order to switch *gie* from logical 0 (current cycle) to 1 (next cycle), the TCB must be executing when *gie* becomes 0 ($\mathbf{X}(PC) \in TCB$)), or the MCU will

reset.

In order to assure that the TCB code is invoked and executed properly, *TAROT* hardware implements LTLs 6.9, 6.10, and 6.11. LTL 6.9 enforces that the only way for the TCB execution to terminate, without causing a *reset*, is through its last instruction (its only legal exit): $PC = TCB_{max}$. This is specified by checking the relation between current and next PC values using LTL ne**X**t operator. If the current PC value is within the TCB, and next PC value is outside the TCB, then either current PC value must be the address of TCB_{max}, or *reset* is set to 1 in the next cycle. Similarly, LTL 6.10 enforces that the only way for PC to enter the TCB is through the very first instruction: TCB_{min}. This prevents the TCB execution from starting at some point in the middle of the TCB, thus making sure that the TCB always executes in its entirety. Finally, LTL 6.11 enforces that *reset* is always set if interrupts or DMA modifications happen during the TCB execution. Even though LTLs 6.9 and 6.10 already enforce that PC can not change to point anywhere outside the TCB, interrupts could be programmed to return to an arbitrary instruction within the TCB. Or, DMA could change $DMEM$ values currently in use by the TCB. Both of these events can alter the TCB behavior and are treated as violations.

Next, Section 6.3.4 presents a computer-checked proof for the sufficiency of this set of sub-properties to imply *TAROT* end-to-end goals. Then, Section 6.3.5 presents FSM-s from our Verilog implementation, that are formally verified to correctly implement each of these requirements.

6.3.4 *TAROT* Composition Proof

TAROT end-to-end sufficiency is stated in Theorems 8 and 9. The complete computer-checked proofs (using Spot2.0 [47]) of Theorems 8 and 9 are publicly available at [4]. Below we present the intuition behind them.

Theorem 8. *Definition 16 ∧ LTLs 6.6,6.7,6.8 → Definition 17.*

Theorem 9. *Definition 16 ∧ LTLs 6.6,6.9,6.10,6.11 → Definition 18.*

Proof of Theorem 8 (Intuition). From machine model's LTL (6.4), as long as the (1) initial trigger configuration is never modified from outside the TCB; and (2) interrupts are never disabled from outside the TCB; it follows that a trigger will cause a proper invocation of the TCB code. Also, successful modifications to the trigger's configuration imply writing to $PMEM$ or IRQ_{cfg} without causing a $reset$ (per LTL (6.2)). Since $TAROT$ verified implementation guarantees that memory modifications to $PMEM$ (LTL (6.6)) or to IRQ_{cfg} (LTL (6.7)) always cause a $reset$, illegal modifications to $trigger_{cfg}$ are never successful. Finally, LTL (6.8) assures that any illegal interrupt disablement always causes a $reset$, and is thus never successful). Therefore, $TAROT$ satisfies all necessary conditions to meet the goal in Definition 17. □

Proof of Theorem 9 (Intuition). The fact that a reset always causes a later call to the TCB follows from the machine model's LTL (6.5) and $TAROT$ guarantee in LTL (6.6). LTLs (6.9) and (6.9) ensure that the TCB executable is properly invoked and executes atomically, until its legal exit. Otherwise a $reset$ flag is set, which (from the above argument) implies a new call to the TCB. Finally, LTL 6.11 assures that any interrupt or DMA activity during the TCB execution will cause a $reset$, thus triggering a future TCB call and satisfying Definition 18. □

See [4] for the formal computer-checked proofs.

6.3.5 Sub-Module Implementation+Verification

We now proceed with the implementation and formal verification of $TAROT$ hardware.

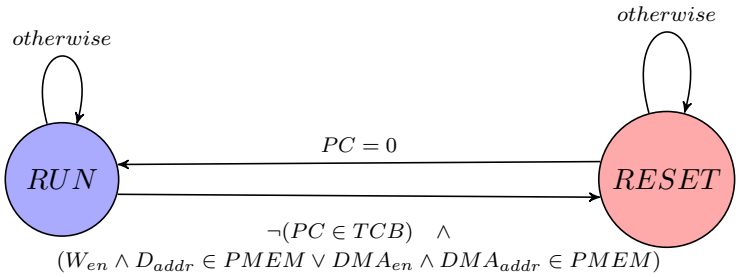


Figure 6.6: Verified FSM for LTL 6.6.

TAROT modules are implemented as Mealy FSMs (where outputs change with the current state and current inputs) in Verilog. Each FSM has one output: a local *reset*. *TAROT* output *reset* is given by the disjunction (logic *or*) of local *reset*-s of all sub-modules. Thus, a violation detected by any sub-module causes *TAROT* to trigger an immediate MCU *reset*. For the sake of easy presentation we do not explicitly represent the value of *reset* in the figures. Instead, we define the following implicit representation:

1. *reset* output is 1 whenever an FSM transitions to the *RESET* state (represented in red color);
2. *reset* output remains 1 until a transition leaving the *RESET* state is triggered;
3. *reset* output is 0 in all other states (represented in blue color).

Note that all FSM-s remain in the *RESET* state until $PC = 0$, which signals that the MCU reset routine finished.

Figure 6.6 illustrates *TAROT* sub-module responsible for assuring that *PMEM* modifications are only allowed from within the TCB. This minimal 2-state machine works by monitoring PC, W_{en}, D_{addr}, DMA_{en}, and DMA_{addr} to detect illegal modification attempts by switching from *RUN* to *RESET* state, upon detection of any such action. It is verified to adhere to LTL (6.6). A similar FSM is used to verifiably enforce LTL (6.7), with the only distinction of checking for writes within IRQ_{cfg} region instead, i.e., $D_{addr} \in IRQ_{cfg}$) and $DMA_{addr} \in IRQ_{cfg}$). Given the similarity, we omit the illustration of this FSM.

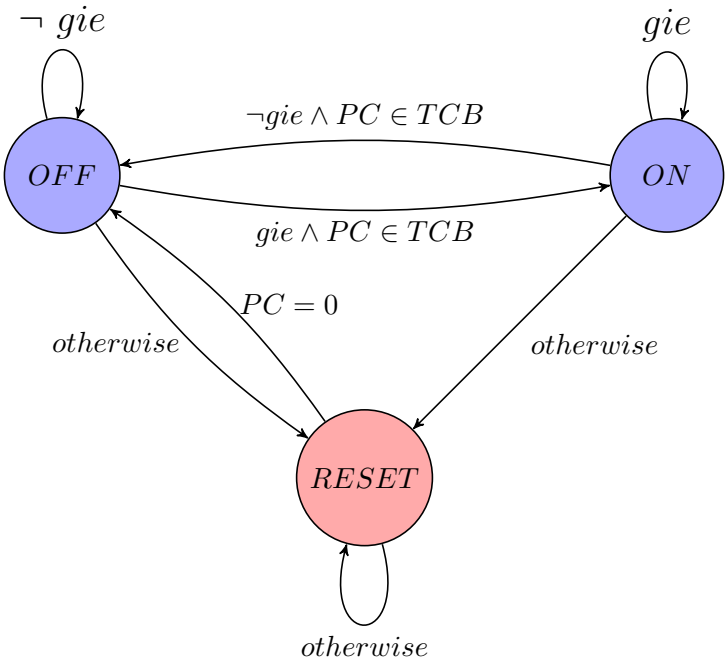


Figure 6.7: Verified FSM for LTL 6.8.

Figure 6.7 presents an FSM implementing LTL 6.8. It monitors the "global interrupt enable" (gie) signal to detect attempts to illegally disable interrupts. It consists of three states: (1) ON, representing execution periods where $gie = 1$; (2) OFF, for cases where $gie = 0$, and (3) $RESET$. To switch between ON and OFF states, this FSM requires $PC \in TCB$, thus preventing misconfiguration by untrusted software.

Finally, the FSM in Figure 6.8 verifiably implements LTLs 6.9, 6.10, and 6.11. This FSM has 5 states, one of which is $RESET$. Two basic states correspond to whenever: the TCB is executing (state "$\in TCB$"), and not executing (state "$\notin TCB$"). From $\notin TCB$ the only reachable path to $\in TCB$ is through state TCB_{entry}, which requires $PC = TCB_{min}$ – TCB only legal entry point. Similarly, from $\in TCB$ the only reachable path to $\notin TCB$ is through state TCB_{exit}, which requires $PC = TCB_{max}$ – TCB only legal exit. Also, in all states where $PC \in TCB$ (including entry and exit transitions) this FSM requires DMA and interrupts to remain inactive. Any violation of these requirements, in any of the four regular states, causes the FSM transition to $RESET$, thus enforcing protection to the TCB execution.

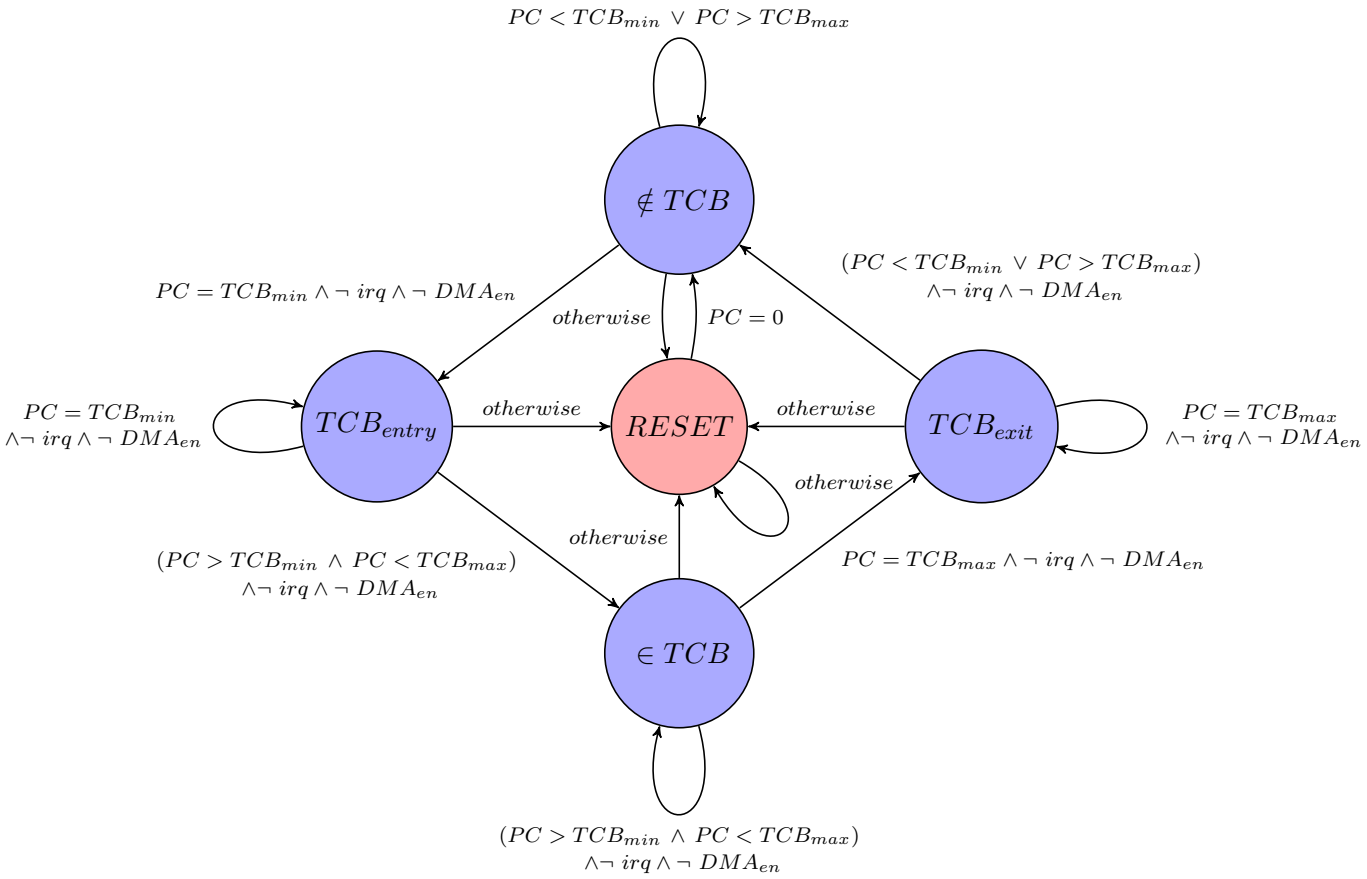


Figure 6.8: Verified FSM for LTLs 6.9–6.11.

6.3.6 TCB Confidentiality

One instance of *TAROT* enables confidentiality of the TCB data and code with respect to untrusted applications. This is of particular interest when $\mathcal{F}$ implements cryptographic functions or privacy-sensitive tasks.

This goal can be achieved by including and epilogue phase in the TCB executable, with the goal of performing a $DMEM$ cleanup, erasing all traces of the TCB execution from the stack and heap. While the TCB execution may be interrupted before the execution of the epilogue phase, such an interruption will cause an MCU *reset*. The *Re-Trigger on Failure* property assures that the TCB code will execute (as a whole) after any *reset* and will thus erase remaining execution traces from $DMEM$ before subsequent execution of untrusted applications. In a similar vein, if confidentiality of the executable is desirable, it can be implemented following LTL (6.12), which formalizes read attempts based on R_{en} signal:

$$\begin{aligned} &\mathbf{G}: \{ \\ &\quad [\neg(PC \in TCB) \wedge R_{en} \wedge (D_{addr} \in TCB) \vee \\ &\quad DMA_{en} \wedge (DMA_{addr} \in TCB)] \rightarrow reset \\ &\} \end{aligned} \tag{6.12}$$

An FSM implementing this property is shown in Figure 6.9. Note that, despite visual similarity with the FSM in Figure 6.6, the confidentiality FSM checks for *reads* (instead of writes) to the TCB (instead of entire $PMEM$).

This property prevents external reads to the TCB executable by monitoring R_{en}, D_{addr}, and DMA. When combined with the aforementioned erasure epilogue, it also enables secure storage of secrets within the TCB binary (as in [9, 10, 48]). This part of *TAROT* design is optional, since some embedded applications do not require confidentiality, e.g., those discussed in Sections 6.4.1 and 6.4.2.

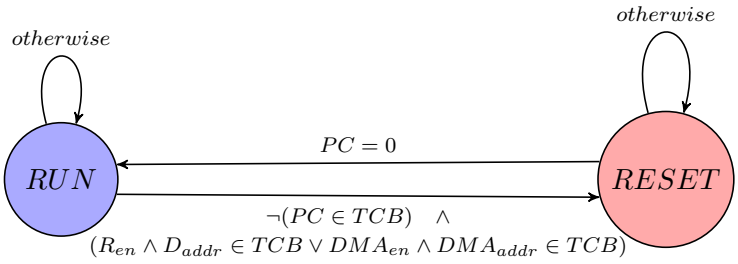


Figure 6.9: Verified FSM for LTL 6.12.

6.3.7 Resets & Availability

One important remaining issue is *availability*. For example, malware might interrupt (or tamper with) with $INIT$ execution after a *reset* preventing the subsequent execution of the TCB. Also, malware could to interrupt the TCB execution, after each *re*-trigger, with the goal of resetting the MCU indefinitely, and thereby preventing the TCB execution from ever completing its task.

We observe that such actions are not possible, since they would require either DMA activity or interrupts to: (1) hijack $INIT$ control-flow; or (2) abuse $TAROT$ to successively *reset* the MCU during the TCB execution after each re-trigger. Given **H5** interrupts are disabled by default at boot time. Additionally, **H4** states that any prior DMA configuration is cleared to the default disabled state after a *reset*. Hence, $INIT$ and the first execution of the TCB after a *reset* cannot be interrupted or tampered with by DMA.

Finally, we note that, despite preventing security violations by (and implementing re-trigger based on) resetting the MCU, $TAROT$ does not provide any advantage to malware that aims to simply disrupt execution of (non-TCB) applications by causing *resets*. Any software running on bare metal (including malware) can always intentionally reset the MCU. Resets are the default mechanism to recover from regular software faults on unmodified (off-the-shelf) low-end MCU-s, regardless of $TAROT$.

```
1  int main() {
2      TCB(0);
3      main_loop();
4      return 0;
5  }
```

Figure 6.10: Program Entry Point

6.4 Sample Applications

Many low-end MCU use-cases and applications can benefit from trigger-based active RoTs. To demonstrate generality of *TAROT*, we prototyped three concrete examples, each with a different type of trigger-s. This section overviews these examples: (1) GPIO-TCB uses external analog events (Section 6.4.1), TimerTCB uses timers (Section 6.4.2), and NetTCB uses network events (Section 6.4.3). Finally, Section 6.4.4 discusses how *TAROT* can match active security services proposed in [115] and [60].

6.4.1 GPIO-TCB: Critical Sensing+Actuation

The first example, GPIO-TCB, operates in the context of a safety-critical temperature sensor. It uses *TAROT* to assure that the sensor's most safety-critical function – *sounding an alarm* – is never prevented from executing due to software compromise of the underlying MCU. We use a standard built-in MCU interrupt, based on General Purpose Input/Output (GPIO) to implement trigger. Since this is our first example, we discuss GPIO-TCB in more detail than the other two.

As shown in Figure 6.10, MCU execution always starts by calling the TCB (at line 2). Therefore, after MCU initialization/reset, unprivileged (non-TCB) applications can only execute after the TCB; assuming, of course, that formal guarantees discussed in Section 6.3 hold. These applications are implemented inside *main_loop* function (at line 3).

The correct trigger configuration in GPIO-TCB can be achieved in two ways. The first way is

```
1  void setup (void) {
2      P1DIR  = 0x00;
3      P1IE   = 0x01;
4      P1IES  = 0x00;
5      P1IFG  = 0x00;
6  }
```

Figure 6.11: Trigger Setup

to set IRQ_{cfg} to the desired parameters at MCU deployment time, by physically writing this configuration to IRQ_{cfg}. The second option is to implement this configuration in software as a part of the TCB. Since the TCB is always the first to run after initialization/reset, it will configure IRQ_{cfg} correctly, enabling subsequent trigger-s at runtime.

Figure 6.11 shows IRQ_{cfg} configuration, implemented as part of the TCB, i.e., called from within the TCB. This `setup` function is statically linked to be located inside the TCB memory region, thus respecting "TCB Execution Protection" LTL rules (see Definition 19). This IRQ_{cfg} setup first configures the physical port $P1$ as an input (line 2, "P1 direction" set to 0x00, whereas 0x01 would set it as an output). At line 3, $P1$ is set as "interrupt-enabled" ($P1IE = 0x01$). A value of $P1IES = 0x00$ (line 4) indicates that, if the physical voltage input of $P1$ changes from logic 0 to 1 ("low-to-high' transition), a GPIO interrupt will be triggered and the respective handler will be called. Finally, $P1IFG$ is cleared to indicate that the MCU is free to receive interrupts (as opposed to busy). We note that this initial trusted configuration of IRQ_{cfg} cannot be modified afterwards by untrusted applications due to $TAROT$ guarantees (see Section 6.3). Based on this configuration, an analog temperature sensing circuit (i.e., a voltage divider implemented using a thermistor (i.e., a resistance thermometer – a resistor whose resistance varies with temperature) is connected to port P1. Resistances in this circuit are set to achieve $5V$ (logic 1) when temperature exceeds a fixed threshold, thus triggering a $P1$ interrupt.

P1 interrupt is handled by the function in Figure 6.12. This is configured using the $interrupt(PORT1\_VECTOR)$ macro. This handler essentially calls $TAROT$ TCB. Parameter 1 in the TCB call distinguishes a regular TCB call from a TCB call following

```
1  interrupt(PORT1_VECTOR) port1_isr(void) {
2      TCB(1);
3  }
```

Figure 6.12: GPIO Handling Routine

```
1  TCB (uint8_t init) {
2       dint();
3       if (!init) {
4           setup();
5       }
6       volatile uint_64 i=0;
7       P3DIR = 0x01;
8       P3OUT = 0x01;
9       while (i<100000000) i++;
10      P3OUT = 0x00;
11      eint();
12      return();
13  }
```

Figure 6.13: IRQ_{cfg} initialization

initialization/reset.

Figure 6.13 depicts the TCB implementation of $\mathcal{F}$. Once triggered, the TCB disables interrupts (`dint`), calls `setup` (if this is the first TCB call after initialization/reset), and activates GPIO port P3 for a predefined number of cycles. P3 is connected to a buzzer (a high frequency oscillator circuit used for generating a buzzing sound), guaranteeing that the alarm will sound. Upon completion, the TCB re-enables interrupts and returns control to the regular application(s).

Note that, as discussed in Section 6.3, executables corresponding to Figures 6.12 and 6.13 are also protected by *TAROT*. Thus, their behavior cannot be modified by untrusted/compromised software.

6.4.2 TimerTCB: Secure Real-Time Scheduling

The second example of *TAROT*, TimerTCB, is in the domain of real-time task scheduling. Without *TAROT* (even in the presence of a passive RoT), a compromised MCU controlled by malware could ignore performing its periodic security- or safety-critical tasks. We show how *TAROT* can ensure that a prescribed task, implemented within the TCB, periodically

```
1  void setup (void) {
2      CCTL0 = CCIE;
3      CCR0  = 1000000;
4      TACTL = TASSEL_2 + MC_1;
5  }
```

Figure 6.14: Timer Trigger Setup

```
1  interrupt(TIMERA0_VECTOR) timera_isr(void) {
2      TCB(1)
3  }
```

Figure 6.15: Timer Handle Routine

executes.

Unlike our first example in Section 6.4.1, TimerTCB only requires modifying IRQ_{cfg}, as illustrated in Figure 6.14. This shows the relative ease of use of *TAROT*. The *setup* function is modified to enable the MCU's built-in timer to cause interrupts (at line 2). Interrupts are set to occur whenever the counter reaches a desired value (at line 3). The timer is set to increment the counter with edges of a particular MCU clock ($MC1$, at line 4). As in the first example, the corresponding interrupt handler is set to always call the TCB (Figure 6.15). In turn, the TCB can implement $\mathcal{F}$ as an arbitrary safety-critical periodic task.

6.4.3 NetTCB: Network Event-based trigger

The last example, NetTCB, uses network event-based trigger to ensure that the TCB quickly filters all received network packets to identify those that carry TCB-destined commands and take action. Incoming packets that do not contain such commands are ignored and passed on to applications through the regular interface (i.e., reading from the UART buffer). In this example, we implement guaranteed receipt of external *reset* commands from some trusted remote entity. This functionality might be desirable after an MCU malfunction (e.g., due to a deadlock) is detected.

In NetTCB, trigger is configured to trap network events. IRQ_{cfg} is set such that each incom-

```
1  void setup (void) {
2      UART_BAUD = BAUD;
3      UART_CTL  = UART_EN | UART_IEN_RX;
4  }
```

Figure 6.16: UART Trigger Setup

```
1  wakeup interrupt (UART_RX_VECTOR) INT_uart_rx(void) {
2      TCB(1);
3  }
4
5  TCB (uint8_t init) {
6      dint();
7      if (!init) {
8          setup();
9      }
10     rxdata = UART_RXD;
11     if (rxdata == 'r') {
12         reset();
13     }
14     eint();
15     return();
16 }
```

Figure 6.17: NetTCB Handler Routine and TCB Implementation

ing UART message causes an interrupt, as shown in Figure 6.16. The TCB implementation, shown in Figure 6.17, filters messages based their initial character $'r'$ which is predefined as a command to *reset* the MCU. **Note that:** in practice such critical commands should be authenticated by the TCB. Although this authentication should be implemented within the TCB, we omit it from this discussion for the sake of simplicity, and refer to [50] for a discussion of authentication of external requests in this setting.

6.4.4 Comparison with [115] and [60]

Recent work proposed security services that can be interpreted as active RoTs. However, these efforts aimed at higher-end embedded devices and require substantial hardware support: Authenticated Watchdog Timer (AWDT) implemented either using a separate (standalone) microprocessor [115], or using ARM TrustZone [60]. Each requirement is, by itself, far more expensive than the cost of a typical low-end MCU targeted in this work (recall the scope discussed in Chapter 2).

In terms of functionality, both [115] and [60] are based on timers. They use AWDT to force

a reset of the device. As in *TAROT* the TCB is the first code to execute; this property is referred to as "gated boot" in [115]. However, unlike *TAROT*, [115, 60] do not consider active RoT behavior obtainable from other types of interrupts, e.g., as in *TAROT* examples in Sections 6.4.1 and 6.4.3. We believe that this is partly because these designs were intended as an active means to enforce memory integrity, rather than a general approach to guaranteed execution of trusted tasks based on arbitrary trigger-s. Note that *TAROT* design is general enough to realize an active means to enforce memory integrity. This can be achieved by incorporating an integrity-ensuring function (e.g, a suitable cryptographic keyed hash) into *TAROT* TCB and using it to check PMEM state upon a timer-based trigger.

Finally, we emphasize that prior results yielded neither formally specified designs nor formally verified implementations. As discussed in Section 6.1, we believe these features to be important for eventual adoption of this type of architecture.

6.5 Implementation & Evaluation

We prototyped *TAROT* (adhering to Figure 6.2) using an open-source implementation of the popular MSP430 MCU – openMPS430 [56] from OpenCores. In addition to *TAROT* module, 2 KBytes of PMEM are reserved for TCB functions. This size choice is configurable at manufacturing time. In our prototype, 2 KBytes is a reasonable choice, corresponding to $5 - 25\%$ of the typical PMEM size in low-end MCU-s. The prototype supports one trigger of each type: timer-based, external hardware, and network. This support is achieved by implementing the IRQ_{cfg} protection, as described in Section 6.3. The MCU already includes multiple timers and GPIO ports that can be selected to act as trigger-s. By default, one of each is used by our prototype. This enables the full set of types of applications discussed in Section 6.4.

As a proof-of-concept, we use Xilinx Vivado to synthesize our design and deploy it using the Basys3 Artix-7 FPGA board. Prototyping using FPGAs is common in both research and industry. Once a hardware design is synthesizable in an FPGA, the same design can be used to manufacture an Application-Specific Integrated Circuit (ASIC) at larger scale.

Hardware & Memory Overhead

Table 6.2 reports *TAROT* hardware overhead as compared to unmodified OpenMSP430 [56]. Similar to the related work [87, 42, 40, 44, 6, 45, 116], we consider hardware overhead in terms of additional Look-Up Tables (LUT-s) and registers. The increase in the number of LUT-s can be used as an estimate of the additional chip cost and size required for combinatorial logic, while the number of registers offers an estimate on the memory overhead required by the sequential logic in *TAROT* FSMs.

TAROT hardware overhead is small with respect to the unmodified MCU core – it requires 2.3% and 4.8% additional LUT-s and registers, respectively. In absolute numbers, *TAROT* adds 33 registers and 42 LUT-s to the underlying MCU.

Runtime & Memory Overhead

We observed no discernible overhead for software execution time on the *TAROT*-enabled MCU. This is expected, since *TAROT* introduces no new instructions or modifications to the MSP430 ISA and to the application executables. *TAROT* hardware runs in parallel with the original MSP430 CPU. Aside from the reserved PMEM space for storing the TCB code, *TAROT* also does not incur any memory overhead. This behavior does not depend on the number of functions or triggers used inside the TCB.

Verification Cost

We verify *TAROT* on an Ubuntu 18.04 machine running at 3.40GHz. Results are also shown

in Table 6.2. *TAROT* implementation verification requires checking 7 LTL statements. The overall verification pipeline is fast enough to run on a commodity desktop in quasi-real-time.

	Hardware		Reserved	Verification			
	Reg	LUT	PMEM/Flash (bytes)	# LTL Invariants	Verified Verilog LoC	Time (s)	Mem (MB)
OpenMSP430 [56]	692	1813	0	-	-	-	-
OpenMSP430 + *TAROT*	725	1855	2048 (default)	7	484	3.1	13.5

Table 6.2: *TAROT* Hardware overhead and verification costs.

Comparison with Prior RoTs

To the best of our knowledge, *TAROT* is the first active RoT targeting this lowest-end class of devices. Nonetheless, to provide a overhead point of reference and a comparison, we contrast *TAROT*'s overhead with that of state-of-the-art passive RoTs in the same class. We note that the results from [115, 60] can not be compared to *TAROT* quantitatively. As noted in Section 6.4.4, [115] relies on a standalone additional MCU and [60] on ARM TrustZone. Both of these are (by themselves) more expensive and sophisticated than the entire MSP430 MCU (and similar low-end MCUs in the same class). Our quantitative comparison focuses on VRASED [40], APEX [42], and SANCUS [87]: passive RoTs implemented on the same MCU and thus directly comparable (cost-wise). Table 6.3 provides a qualitative comparison between the aforementioned relevant designs. Figure 6.18 depicts the relative overhead (in %) of *TAROT*, VRASED, APEX, and SANCUS with respect to the total hardware cost of the unmodified MSP430 MCU core.

In comparison with prior passive architectures, *TAROT* presents lower hardware overhead. In part, this is due to the fact that it leverages interrupt hardware support already present in the underlying MCU to implement its triggers. SANCUS presents substantially higher cost as it implements task isolation and a cryptographic hash engine (for the purpose of verifying software integrity) in hardware. VRASED presents slightly higher cost than *TAROT*. It also necessitates some properties that are similar to *TAROT*'s (e.g., access control to particular memory segments and atomicity of its attestation implementation). In addition, VRASED

Architecture	Behavior	Service	HW Support	Verified?
VRASED [40]	Passive	Attestation	RTL Design	Yes
SANCUS [87]	Passive	Attestation & Isolation	RTL Design	No
APEX [42]	Passive	Attestation & Proof of Execution	RTL Design	Yes
Cider [115]	Active	Timer-based trigger	Additional MCU	No
Lazarus[60]	Active	Timer-based trigger	ARM TrustZone	No
TAROT (this work)	Active	IRQ-based trigger	RTL Design	Yes

Table 6.3: Qualitative Comparison

also requires hardware support for an exclusive stack in DMEM. APEX hardware is a superset of VRASED's, providing an additional proof of execution function in hardware. As such it requires strictly more hardware support, presenting slightly higher cost. *TAROT* also reserves approximately 3.1% (2 KBytes) of the MCU-s 16-bit address space for storing the TCB code. This value is freely configurable, and chosen as a sensible default to support our envisioned RoT tasks (including sample applications in Section 6.4). *TAROT*-enabled MCUs manufactured for different use-cases could increase or decrease this amount accordingly.

6.6 Related Work

Aside from closely related work in [115] and [60] (already discussed in Section 6.4.4), several efforts yielded *passive* RoT designs for resource-constrained low-end devices, along with formal specifications, formal verification and provable security.

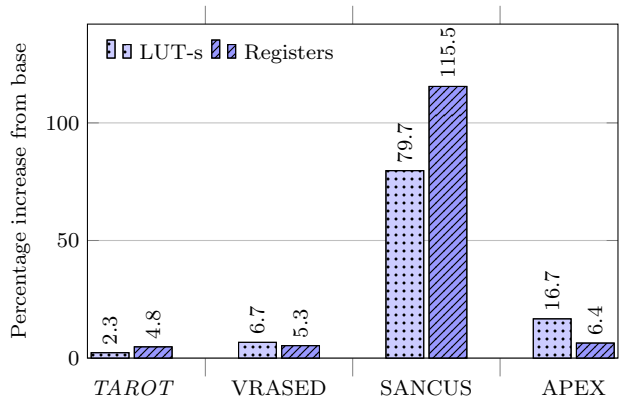


Figure 6.18: Comparison with passive RoTs: Hardware overhead

Low-end RoTs fall into three general categories: software-based, hardware-based, or hybrid. Establishment of software-based RoTs [66, 99, 100, 101, 54, 77, 57] relies on strong assumptions about precise timing and constant communication delays, which can be unrealistic in the IoT ecosystem. However, software-based RoTs are the only viable choice for legacy devices that have no security-relevant hardware support. Hardware-based methods [93, 109, 72, 96, 84, 82, 87] rely on security provided by dedicated hardware components (e.g., TPM [109] or ARM TrustZone [14]). However, the cost of such hardware is normally prohibitive for lower-end IoT devices. Hybrid RoTs [49, 42, 40, 22, 68] aim to achieve security equivalent to hardware-based mechanisms, yet with lower hardware cost. They leverage minimal hardware support while relying on software to reduce the complexity of additional hardware.

In terms of functionality, such embedded RoTs are passive. Upon receiving a request from an external trusted *Verifier*, they can generate unforgeable proofs for the state of the MCU or that certain actions were performed by the MCU. Security services implemented by passive RoTs include: (1) memory integrity verification, i.e., remote attestation [49, 87, 40, 10, 22, 68]; (2) verification of runtime properties, including control-flow and data-flow attestation [84, 42, 44, 6, 45, 116, 105, 43, 55]; as well as (3) proofs of remote software updates, memory erasure, and system-wide resets [41, 9, 15]. As discussed in Section 6.1 and demonstrated in Section 6.4, several application domains and use-cases could greatly benefit from more active RoTs. Therefore, the key motivation for *TAROT* is to not only provide proofs that actions have been performed (if indeed they were), but also to assure that these actions will necessarily occur.

Formalization and formal verification of RoTs for MCU-s is a topic that has recently attracted lots of attention due to the benefits discussed in Chapter 2 and Section 6.1. VRASED [40] (Chapter 3) implemented the first formally verified hybrid remote attestation scheme. APEX [42] (Chapter 5) builds atop VRASED to implement and formally verify

an architecture that enables proofs of remote execution of attested software. PURE [41] implements provably secure services for software updates, memory erasure, and system-wide resets atop VRASED RoT. Another recent result [24] formalized, and proved security of, a hardware-assisted mechanism to prevent leakage of secrets through time-based side-channel that can be abused by malware in control of the MCU interrupts. Inline with aforementioned work, *TAROT* also formalizes its assumptions along with its goals and implements the first formally verified active RoT design.

6.7 Conclusions

This chapter motivated and illustrated the design of *TAROT*: an active RoT targeting low-end MCU-s used as platforms for embedded/IoT/CPS devices that perform safety-critical sensing and actuation tasks. We believe that *TAROT* is the first clean-slate design of a active RoT and the first one applicable to lowest-end MCU-s, which cannot host more sophisticated security components, such as ARM TrustZone, Intel SGX or TPM-s. We believe that this work is also the first formal treatment of the matter and the first active RoT to support a wide range of RoT trigger-s.

Chapter 7

Final Remarks

This dissertation tackles open problems in software integrity and availability in resource-constrained embedded systems that, despite performing safety-critical tasks, can not avail themselves of sophisticated security mechanisms. To this end, we proposed four low-cost security architectures: *VRASED*, *RATA*, *APEX*, and *TAROT*. Our designs focused on simplicity and minimality via hardware/software co-design methodology. This approach made our designs practical (from a cost standpoint) and deployable in the simplest MCUs, that are commonly found at the edge of complex CPS, implementing sensing and actuation tasks. They were also based upon systematic and formally verified properties (and provable composition thereof). We believe this systematic treatment significantly increases the robustness of these architectures. We hope that this first step can inspire future designs of new security services with formal guarantees, potentially also targeting more complex devices.

VRASED, described in Chapter 3, is the first hybrid RA architecture to offer formally verified guarantees for both software and hardware components. It also includes a formal definition of RA security (in the form of a security game) and proofs for the composition of individual sub-modules. Furthermore, it established a verification pipeline, based on LTL model checking

and theorem proving, that has been used to proof security of architectures described in the following chapters.

RATA, presented in Chapter 4, is the first TOCTOU-Secure hybrid RA architecture. It guarantees that even transient malware is detectable. In most cases, *RATA* greatly reduces overall RA runtime overhead from linear to constant time in the size of attested memory. As discussed in Chapter 4, *RATA* also benefits RA in real-time settings, collective RA protocols, and runtime attestation (e.g., control-flow and data-flow attestation).

APEX, described in Chapter 5, augmented RA with verified capabilities to prove execution properties, in addition to memory contents. It enables several important applications – e.g., trustworthy sensing and actuation – that are crucial to the robustness of safety-critical CPS. Subsequent work [43, 91] also demonstrated that *APEX* is sufficient, as the sole hardware support, to enable control-flow attestation and data-flow attestation. In turn, these enable detection of software exploits due to vulnerable implementations of critical operations in low-end MCUs.

Finally, *TAROT* (Chapter 6) focused on the availability dimension of low-end embedded system security. We designed, implemented, and verified an architecture capable of guaranteeing that specific safety-critical tasks are always executed upon particular trigger-s, despite full compromise of all other software in the MCU.

Based on the lessons learned in this work, several avenues remain for future work. Aside from the natural direction of addressing current limitations of proposed architectures, we foresee the following general directions: (i) design and verification of similar security services for slightly higher-end embedded devices, e.g., equipped with rudimentary privilege/exception levels and MPUs; (ii) design of trusted architectures to solve other problems arising from low-end sensors and actuators, such as sensing privacy; and (iii) deployment and interoperability of proposed technologies as a part of larger-scale CPS and cryptographic protocols.

Bibliography

[1] VRASED source code. https://github.com/sprout-uci/vrased, 2019.

[2] APEX source code. https://github.com/sprout-uci/APEX, 2020.

[3] *TAROT* source code. https://www.dropbox.com/sh/y7nwc9y3bc4z9u7/AACwMGGK3uxKqiCSWYqpm1iVa?dl=0, 2021.

[4] RATA source code (to be moved to github soon). https://www.dropbox.com/sh/qs8ylburhtp7tzc/AABdaqjOL5lUtZy2nt51E1Jza?dl=0, 2021.

[5] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A.-R. Sadeghi, and G. Tsudik. Things, trouble, trust: on building trust in iot systems. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[6] T. Abera et al. C-flat: Control-flow attestation for embedded systems software. In *CCS '16*, 2016.

[7] E. Aliaj, I. D. O. Nunes, and G. Tsudik. Garota: Generalized active root-of-trust architecture. *arXiv preprint arXiv:2102.07014*, 2021.

[8] M. Ambrosin et al. SANA: Secure and Scalable Aggregate Network Attestation. In *CCS*, 2016.

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

[10] M. Ammar, B. Crispo, and G. Tsudik. Simple: A remote attestation approach for resource-constrained iot devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

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

[12] 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.

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

[14] Arm Ltd. Arm TrustZone. `https://www.arm.com/products/security-on-arm/trustzone`, 2018.

[15] 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.

[16] N. Asokan et al. Seda: Scalable embedded device attestation. In *ACM CCS*, 2015.

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

[18] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, 2012.

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

[20] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede. Spongent: The design space of lightweight cryptographic hashing. *IEEE Transactions on Computers*, 62, 2013.

[21] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX*, 2017.

[22] F. Brasser et al. Tytan: Tiny trust anchor for tiny devices. In *DAC*, 2015.

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

[24] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 262–276. IEEE, 2020.

[25] G. Cabodi, P. Camurati, S. F. Finocchiaro, C. Loiacono, F. Savarese, and D. Vendraminetto. Secure embedded architectures: Taint properties verification. In *DAS*, 2016.

[26] G. Cabodi, P. Camurati, C. Loiacono, G. Pipitone, F. Savarese, and D. Vendraminetto. Formal verification of embedded systems for remote attestation. *WSEAS Transactions on Computers*, 14, 2015.

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

[28] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik. Lightweight swarm attestation: a tale of two lisa-s. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 86–100. ACM, 2017.

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

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

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

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

[33] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[34] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

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

[36] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[37] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[38] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.

[39] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *IEEE DISCEX*. IEEE, 2000.

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

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

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

[43] I. De Oliveria Nunes, S. Jakkamsetti, and G. Tsudik. Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution. In *Design, Automation and Test in Europe Conference (DATE)*, 2021.

[44] 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)*, pages 1–8. IEEE, 2018.

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

[46] X. Du and H.-H. Chen. Security in wireless sensor networks. *IEEE Wireless Communications*, 15(4):60–66, 2008.

[47] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0a framework for ltl and ω-automata manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, 2016.

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

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

[50] B. et al. Remote attestation for low-end embedded devices: the prover's perspective. In *DAC*, 2016.

[51] G. A. Fowler. Alexa has been eavesdropping on you this whole time. `https://www.washingtonpost.com/technology/2019/05/06/alexa-has-been-eavesdropping-you-this-whole-time/`, 2019.

[52] A. Francillon et al. A minimalist approach to remote attestation. In *DATE*, 2014.

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

[54] R. W. Gardner, S. Garera, and A. D. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE TIFS*, 2009.

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

[56] O. Girard. openMSP430, 2009.

[57] V. D. Gligor and S. L. M. Woo. Establishing software root of trust unconditionally. In *NDSS*, 2019.

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

[59] G. Hinterwälder, A. Moradi, M. Hutter, P. Schwabe, and C. Paar. Full-size high-security ECC implementation on MSP430 microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 31–47. Springer, 2014.

[60] M. Huber, S. Hristozov, S. Ott, V. Sarafov, and M. Peinado. The lazarus effect: Healing compromised devices in the internet of small things. *arXiv preprint arXiv:2005.09714*, 2020.

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

[62] A. Ibrahim et al. DARPA: Device Attestation Resilient against Physical Attacks. In *WiSec*, 2016.

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

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

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

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

[67] G. Klein, K. Elphinstone, G. Heiser, et al. seL4: Formal verification of an OS kernel. In *ACM SIGOPS*, 2009.

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

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

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

[71] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 2012.

[72] X. Kovah et al. New results for timing-based attestation. In *IEEE S&P '12*, 2012.

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

[74] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[75] Y. Li, Y. Cheng, V. Gligor, and A. Perrig. Establishing software-only root of trust on embedded systems: Facts and fiction. In *Security Protocols—22nd International Workshop*, 2015.

[76] Y. Li, J. M. McCune, and A. Perrig. Viper: Verifying the integrity of peripherals' firmware. In *CCS*. ACM, 2011.

[77] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *ACM CCS*, 2011.

[78] Y. Lindell and J. Katz. *Introduction to modern cryptography*, chapter 4.3, pages 109–113. Chapman and Hall/CRC, 2014.

[79] D. W. Loveland. *Automated Theorem Proving: a logical basis*. Elsevier, 2016.

[80] F. Lugou, L. Apvrille, and A. Francillon. Toward a methodology for unified verification of hardware/software co-designs. *Journal of Cryptographic Engineering*, 2016.

[81] F. Lugou, L. Apvrille, and A. Francillon. Smashup: a toolchain for unified verification of hardware/software co-designs. *Journal of Cryptographic Engineering*, 7(1):63–74, 2017.

[82] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P '10*, 2010.

[83] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[84] J. McCune et al. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Operating Systems Review*, 2008.

[85] K. L. McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[86] L. Narula and T. E. Humphreys. Requirements for secure clock synchronization. *IEEE Journal of Selected Topics in Signal Processing*, 12(4):749–762, 2018.

[87] J. Noorman, J. V. Bulck, J. T. Mühlberg, et al. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3), 2017.

[88] J. Noorman et al. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX*, 2013.

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

[90] I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik. On the toctou problem in remote attestation. *arXiv preprint arXiv:2005.03873*, 2020.

[91] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik. Dialed: Data integrity attestation for low-end embedded devices. 2021.

[92] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, 2010.

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

[94] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, et al. Verified low-level programming embedded in f. *Proceedings of the ACM on Programming Languages*, 2017.

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

[96] D. Schellekens et al. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 2008.

[97] 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*, 2006.

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

[99] A. Seshadri et al. SWATT: Software-based attestation for embedded devices. In *IEEE S&P '04*, 2004.

[100] A. Seshadri et al. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[101] A. Seshadri et al. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS*. 2008.

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

[103] L. Simon, D. Chisnall, and R. Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. In *Proceedings of the Third IEEE European Symposium on Security and Privacy (EuroSP)*, London, UK, Apr. 2018. ACM SIGOPS.

[104] R. V. Steiner and E. Lupu. Attestation in wireless sensor networks: A survey. *ACM Computing Surveys (CSUR)*, 49(3):51, 2016.

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

[106] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[107] Texas Instruments. MSP430 GCC user's guide, 2016.

[108] J. Torrellas. Architectures for extreme-scale computing. *Computer*, 42(11):28–35, 2009.

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

[110] G. S. Tuncay, S. Demetriou, K. Ganju, and C. A. Gunter. Resolving the predicament of Android custom permissions. In *ISOC Network and Distributed Systems Security Symposium (NDSS)*, 2018.

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

[112] A. Virtualization. Secure virtual machine architecture reference manual. *AMD Publication*, 33047, 2005.

[113] Xilinx. Vivado design suite user guide, 2017.

[114] Xilinx Inc. Artix-7 FPGA family, 2018.

[115] M. Xu, M. Huber, Z. Sun, P. England, M. Peinado, S. Lee, A. Marochko, D. Mattoon, R. Spiger, and S. Thom. Dominance as a new trusted computing primitive for the internet of things. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1415–1430. IEEE, 2019.

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

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