UNIVERSITY OF CALIFORNIA,
IRVINE

Root-of-Trust Architectures for Low-end Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Sashidhar Jakkamsetti

Dissertation Committee:
Professor Gene Tsudik, Chair
Professor Alfred Chen
Professor Ardalan Amiri Sani

2023

DEDICATION

To my beloved family – my parents, Padmasree and Someswara Rao, my wife, Apoorva, and my brother, Venkatesh.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Such a wonderful five years of my life! Ph.D. was indeed a monumental learning experience. It taught me patience and diligence and helped me evolve into a better and more mature person. I am truly happy that I pursued this journey, and I am deeply grateful to everyone who played a role, whether knowingly or unknowingly, in making it remarkable.

Above all, I am incredibly thankful to my amazing advisor, Gene Tsudik, for providing me with the opportunity to pursue this Ph.D. with his generous support and guidance. He taught me how to conduct proper research and, especially, how to write scientific papers. His simple and unique style of writing and presenting research makes things easy to understand, and I am glad that I could take at least some of it with me.

I extend my sincere thanks to my committee members, Ardalan Amiri Sani and Alfred Chen, for their interest in my research and their valuable feedback and expertise. Having such exceptional researchers on my Ph.D. committee was truly a privilege.

I also owe my gratitude to my early mentors without whom this journey wouldn't have started. First of all, I would like to convey my sincere thanks to Jean-Pierre Seifert, my mentor during my first internship while pursuing Bachelor's degree. He is one of the reasons for my early interest in security. I would also like to thank my Bachelor's advisors – Debashis Ghosh and Vinod Pankajakshan – who were the very first to introduce me to research.

In addition to the mentors who helped me career-wise, I would also like to express my immense respect and gratitude towards my early teachers, Krishnan Pagalthivarthi and Ankush Mittal, who inspired me to pursue a Ph.D. and taught me how to lead a happy life.

Coming to my extraordinary collaborators during my Ph.D., I want to extend my heartfelt thanks to all whom I have worked with, am currently working with, and/or continue to work with: Ivan De Oliveira Nunes, Benjamin Terner, Seoyeon Hwang, Norrathep Rattanavipanon, Youngil Kim, Karim Eldefrawy, Moti Yung, Zeyu Liu, and Varun Madathil. Some of them were also my fellow labmates at UCI who were directly involved in some parts of this dissertation. I especially want to mention Ivan De Oliveira Nunes, my close mentor, with whom I had a lot of fun working, discussing, sharing ideas, and writing papers together. I have learned a lot from him. I also want to specifically thank Benjamin Terner and Karim Eldefrawy for introducing me to other interesting topics (apart from this dissertation) during one of my internships, which I would like to further explore. This list of collaborators is never-ending... thank you all so so much!

It would be incomplete without mentioning my dear friends that I made at UCI. I am very thankful to all my labmates who were there for me throughout my journey: Norrathep Rattanavipanon, Ivan De Oliveira Nunes, Ercan Ozturk, Yoshimichi Nataksuka, Seoyeon Hwang, Andrew Searles, Youngil Kim, Renascence Tarafder Prapty, Elina Van Kempen, and Gene Tsudik. Here's to all the fun times!

Last but not least, my delightful family and friends. This phase of my life wouldn't have

been this great if it weren't for all of you. First and foremost, I want to express my deep gratitude to my loving, understanding, and caring wife, Apoorva Muthineni. She was the reason I made it this far. I can't thank my parents, Padmasree and Someswara Rao Jakkamsetti, and my brother, Venkatesh Jakkamsetti, enough. You guys have been my strongest supporters throughout my life; this dissertation belongs to you! I would also like to thank my grandparents, uncles, aunts, and cousins in my family. Additionally, I am very grateful to all my friends in the US and India for their love, laughter, and support.

—

Portion of Chapter 3 is a reprint of the material as it appears in the Proceedings of 24th Design, Automation and Test in Europe Conference (DATE 2021) [53], used with permission from the Institute of Electrical and Electronics Engineers.

Portion of Chapter 4 is a reprint of the material as it appears in the Proceedings of the 58th Design Automation Conference (DAC 2021) [52], used with permission from the Institute of Electrical and Electronics Engineers.

Portion of Chapter 5 is a reprint of the material as it appears in the Proceedings of 43rd IEEE Symposium on Security and Privacy (S&P 2022) [104], used with permission from the Institute of Electrical and Electronics Engineers.

Portion of Chapter 6 is a reprint of the material as it appears in the Proceedings of the 41st International Conference on Computer-Aided Design (ICCAD 2022) [50], used with permission from the Association for Computing Machinery.

Financial support was provided by Gene Tsudik (via Graduate Student Researcher position), University of California, Irvine, UCI School of ICS (via Teaching Assistant position); and other grants and funding sources (via Gene): Army Research Office under contract W911NF-16-1-0536, Semiconductor Research Corporation under contract 2019-TS-2907, NSF Awards 1956393-SATC and 1840197-CICI, DARPA subcontract from Peraton Labs.

VITA

Sashidhar Jakkamsetti

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, California*

Bachelor of Technology in Electronics & Communications Eng. **2016**
Indian Institute of Technology, Roorkee *Roorkee, India*

PROFESSIONAL EXPERIENCE

Research Scientist **2023 – present**
Bosch Research *Sunnyvale, California*

Research Engineer Intern **Summer 2022**
Meta Research *Menlo Park, California*

Ph.D. Research Intern **Summer 2021**
Visa Research *Menlo Park, California*

Security Research Intern **Summer 2019 & 2020**
SRI International *Menlo Park, California*

Software Development Engineer **2016 – 2018**
Microsoft *Hyderabad, India*

Research Intern & DAAD Scholar **Summer 2015**
Deutsche Telekom & Technical University of Berlin *Berlin, Germany*

TEACHING EXPERIENCE

Teaching Assistant **2018 – 2019**
University of California, Irvine *Irvine, California*

PAPERS IN SUBMISSION OR UNDER REVIEW*

Ivan De Oliveira Nunes, Seoyeon Hwang, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. **PARseL: Towards a Verified Root-of-Trust over seL4.** In submission at *International Conference on Computer-Aided Design (ICCAD)*, 2023.

Sashidhar Jakkamsetti, Youngil Kim, and Gene Tsudik. **Caveat (IoT) Emptor: Towards Transparency of IoT Device Presence.** In submission at *ACM Conference on Computer and Communications Security (CCS)*, 2023.

Sashidhar Jakkamsetti, Zeyu Liu, and Varun Madathil. **Scalable Private Signaling.** In submission at *IEEE Symposium on Security and Privacy (S&P)*, 2023.

Karim Eldefrawy, Sashidhar Jakkamsetti, Ben Terner, and Moti Yung. **Standard Model Time-Lock Puzzles: Defining Security and Constructing via Composition.** In *IACR Cryptology ePrint Archive*, 2023.

REFERRED CONFERENCE PUBLICATIONS*

Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Youngil Kim, and Gene Tsudik. **CASU: Compromise Avoidance via Secure Updates for Low-end Embedded Systems.** Appeared at *International Conference on Computer-Aided Design (ICCAD)*, 2022.

Ivan De Oliveira Nunes, Seoyeon Hwang, Sashidhar Jakkamsetti, and Gene Tsudik. **Privacy-from-Birth: Protecting Sensed Data from Malicious Sensors with VERSA.** Appeared at *IEEE Symposium on Security and Privacy (S&P)*, 2022.

Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. **On the TOCTOU Problem in Remote Attestation.** Appeared at *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.** Appeared at *Design Automation Conference (DAC)*, 2021.

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

***Authors in the listed publications are arranged in alphabetical order of their last name.**

OTHER PUBLICATIONS

Sugareddy, Sai Rohith, Sashidhar Jakkamsetti, Ramesh Goud, Yashaswini K.V. **Digi Analysis: Static And Dynamic Evaluation Of Facial Asymmetry & Mandibular Deviation**. Appeared at *International Journal of Current Advanced Research*, 2021. (*Runner-up in Competitive Table Clinic (Research)*, Indian Orthodontic Society, 2021.)

OPEN-SOURCE CONTRIBUTIONS

Scalable Private Signaling `https://github.com/sashidhar-jakkamsetti/sgx-ps`
Implementation of Scalable Private Signaling using Intel SGX SDK and Drivers, and OpenSSL written in C++.

CASU `https://github.com/sprout-uci/CASU`
Implementation of CASU using Xilinx Vivado, Diligent Basys3 FGPA, OpenMSP430, and HACL written in Verilog and C.*

VERSA `https://github.com/sprout-uci/pfb`
Implementation of VERSA using Xilinx Vivado, Diligent Basys3 FGPA, OpenMSP430, and HACL written in Verilog and C.*

VRASED+ `https://github.com/sprout-uci/vrased-plus`
Implementation of VRASED+ (VRASED with Verifier Authentication) using Xilinx Vivado, Diligent Basys3 FGPA, OpenMSP430, and HACL written in Verilog and C.*

RATA `https://github.com/sprout-uci/RATA`
Implementation of RATA using Xilinx Vivado, Diligent Basys3 FGPA, OpenMSP430, and HACL written in Verilog and C.*

ABSTRACT OF THE DISSERTATION

Root-of-Trust Architectures for Low-end Embedded Systems

By

Sashidhar Jakkamsetti

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Gene Tsudik, Chair

Internet-of-Things (IoT), "smart", and Cyber-Physical Systems (CPS) devices have become increasingly popular and commonplace over the past two decades. Some of them perform safety-critical tasks and collect sensitive information, e.g., smoke detectors, temperature sensors, heart rate monitors, and fitness trackers. However, due to their stringent cost, size, and energy constraints, they are equipped with few (or no) security features. This makes them vulnerable to attacks. Some prior work proposed security architectures (such as remote attestation, proof of execution, and secure reset/erasure) to detect and mitigate malware on them. However, these approaches either partially mitigate the problem and/or require new hardware that is unrealistic for low-end devices.

This dissertation presents four hybrid (hardware/software co-design) root-of-trust (RoT) architectures that mitigate various attacks with small hardware modifications: TinyCFA, DIALED, VERSA, and CASU. TinyCFA and DIALED are passive RoTs that detect runtime (control-flow and data-only) attacks by proposing new control-flow and data-flow attestation architectures respectively. Whereas, VERSA and CASU are active RoTs that prevent sensor data privacy leakage and code-injection attacks based on hardware-enforced access control mechanisms. We implement and evaluate these architectures on low-end microcontrollers (e.g., TI MSP430) and show that they are suitable for resource-constrained IoT. We also

formally verify the hardware implementation of VERSA and CASU, thus showing that they meet all stated security requirements.

Chapter 1

Introduction

Internet-of-Things (IoT) and Cyber-Physical Systems (CPS) refer to (inter)connected embedded devices equipped with sensors, actuators, control units, and network connectivity that enable them to collect, process, and exchange data. Such special-purpose computing devices have become increasingly ubiquitous and widely adopted in many everyday settings, including both private (e.g., homes, offices, and factories) and public (e.g., cultural, entertainment, and transportation) spaces; they are also frequently used in farming, industrial, and vehicular automation. This enables the creation of "smart" environments that improve efficiency, productivity, and convenience in various aspects of modern society. It is expected that the number of IoT devices will exceed 29 billion by 2030 [122], indicating their continued growth and important role in shaping the future of industries, societies, and economies.

In contrast with general-purpose computers, IoT devices are purpose-built for specific tasks that involve sensing and/or actuation. Some of these devices have safety-critical functions, such as smoke/fire detectors, smart door locks, surveillance cameras, and other safety sensors used in industrial and automobile settings. On the other hand, there are IoT devices that gather and process sensitive personal information, including wearables like fitness trackers,

smartwatches, and health monitors such as pacemakers, connected inhalers, and syringe pumps.

IoT device manufacturers understandably prioritize the development of novel functionality, external aesthetics, user-friendliness, and other factors. Unfortunately, security is often considered a secondary concern or an afterthought. This is partly due to various constraints such as physical space, energy consumption, and cost considerations.

Unsurprisingly, IoT devices have been easy and attractive attack targets over the past few years, e.g., [121, 94, 58, 42]. In 2010, Stuxnet Worm [121] targeted industrial control systems (specifically, programmable logic controllers) used in Iran's nuclear facilities. By exploiting vulnerabilities in their firmware, Stuxnet disrupted centrifuge operations, causing physical damage to the equipment and sabotaging the nuclear enrichment process. In 2016, Mirai Botnet [94] launched a large-scale Distributed Denial-of-Service (DDoS) attack that infected a vast number of poorly secured IoT devices, including cameras and routers. Mirai also zombified Dyn, a major DNS provider in the US, causing widespread service disruptions and making them inaccessible for several hours. There are also many Mirai-inspried Botnets (e.g. Bashlite [7] and Hajime [8]) that search for, and infect, vulnerable IoT devices in order to later take advantage of them. In 2017, Triton (or Trisis) malware [58] attacked industrial safety systems used in oil and gas related infrastructures, leading to physical accidents and disruptions. There are also numerous attacks targeting IoT user privacy [63, 64, 130, 113, 16]. Notably, [130, 16] attacked smart home devices and eavesdropped on users to obtain sensitive private information.

A lot of research effort was invested in mitigating security and privacy issues in the IoT ecosystem. Many proposed techniques (e.g., [132, 78, 91]) use end-to-end encryption, authentication, and other cryptographic constructs to secure IoT devices and their communication with cloud servers. Another research direction focused on protecting sensitive data from passive in-network adversaries, (e.g. [127, 22, 23]) and performing analysis based on traffic

metadata, e.g., packet header fields, size, and frequency. However, most of the prior work assumes that these devices execute expected benign software and thus perform their expected functionality, including cryptographic operations upon which their security generally relies.

Aforementioned techniques are suitable for high-end computing devices, such as smartphones and laptops, because they are, by default, equipped with protective hardware such as Memory Protection Units (MPUs), Memory Management Units (MMUs), or Trusted Execution Environments (TEEs). Such devices also support operating systems or microkernels to provide virtualization and isolate security-sensitive functions. However, the situation becomes more challenging with low-power and budget-constrained embedded devices.

Low-end embedded devices are typically equipped with one or more resource-constrained micro-controller unit(s) (MCUs) (for example, TI MSP430 [74] and AVR AtMega32 [5]), which cannot run complex software or host expensive hardware. They often run simple software on "bare metal" with minimal (or no) security mechanisms in place. Consequently, it is impractical to assume that these MCUs, by themselves, can mitigate attacks, unlike their higher-end counterparts.

A naive approach to protect these MCUs is by making all software to be read and execute only. While this idea prevents malware from tampering with the software, it also sacrifices flexibility by precluding any software updates.

Motivated by the aforementioned concerns, several small Root-of-Trusts (RoTs) were recently proposed to provide rudimentary security services such as remote attestation [62, 47, 119, 21, 81, 117], proof of remote software execution [49, 118], proof of secure reset, erase, and update [27, 115, 48] on low-end devices. These tiny RoTs are usually implemented as hardware/software (hybrid) co-designs, aiming to achieve similar security levels as more costly hardware-based architectures, albeit, at much lower hardware cost. See Chapter 2 for more details on low-end MCUs and hybrid RoTs.

Nonetheless, current state-of-the-art can only measure software integrity, which although useful, is not sufficient for a comprehensive security design. Their key limitations are:

1. Techniques such as remote attestation and proofs of software execution measure software integrity during runtime. However, this does not capture the control-flow path taken by the software. For example, a buffer overflow attack can corrupt the stack or heap, potentially leading to a *control-flow attack*, i.e., where the software takes an unexpected (or unintended) execution path. In that case, despite measuring software integrity, the result of the execution is incorrect.

2. *Data-only attacks* target and corrupt the critical data variables while allowing the software to follow expected control-flow path, thus remaining undetected. Techniques that mitigate such attacks in sophisticated devices are not applicable to low-end MCUs since they require much additional hardware.

3. Most prior results are *reactive* in nature, meaning they can detect and report software compromises after the fact. However, they cannot prevent such compromises. Merely detecting software compromises is not very useful in privacy-sensitive or safety-critical applications. For example, consider a smoke detector. If the malware launches a *code injection attack*, it can prevent the alarm from sounding even in the presence of smoke. In such a scenario, despite attestation detecting the compromise, the consequences could be catastrophic. Moreover, attestation during such real-time applications is a hindrance because of its runtime overhead.

This dissertation explores new techniques in order to fill the gap in securing low-end resource-constrained MCUs.

1.1 Contributions

We present four RoT architectures for low-end MCUs:

1. *Tiny-CFA* – a control-flow attestation architecture that detects control-flow attacks by measuring the control-flow path taken by the software during runtime.
2. *DIALED* – a data-flow attestation architecture that detects both control-flow and data-only attacks by measuring data-flow taken by the software during runtime.
3. *VERSA* – a privacy architecture that prevents sensor data leakage by enabling a hardware-enforced access control mechanism.
4. *CASU* – a security architecture that prevents code-injection attacks by enforcing benign software immutability and enabling secure authorized software updates.

The first two are *passive* architectures that detect and report runtime software exploits in low-end IoT devices. Whereas, the second two are *active* architectures that prevent privacy leakage and code modification attacks. They are specifically targeted toward personal or safety-critical devices.

The focus of this dissertation is on designing, implementing, and evaluating these RoTs. Another contribution of this dissertation is it implements formally verified hardware to derive its security properties. For the first two RoTs, we use verified hardware to extend runtime execution integrity, while for the other two, we implement new hardware, and verify it, to enforce the stated security properties.

Additional hardware used in the proposed architectures is kept minimal, accounting for less than 10-15% of the unmodified MCU core, making it suitable for low-end devices. Furthermore, all source code (including verification proofs) of *VERSA* and *CASU* are open-sourced.

1.2 Scope and Limitations

As usual with most new techniques, those proposed in this dissertation have limitations with respect to applicability, deployability, and security:

1. In terms of scope, they only apply to very low-end MCUs that perform simple tasks. Hence, they are probably not suitable for complex devices that run sophisticated software, as they may incur noticeable overhead when the software size increases. However, in our experiments (see the following Chapters), the overhead incurred by proposed RoTs is tolerable.

2. With regard to deployability, proposed techniques are not applicable to off-the-shelf MCUs, due to reliance on custom hardware support. However, all proposed techniques can be extended to MCUs equipped with hardware support, such as a TEE (e.g. ARM TrustZone [26]) with minor modifications.

3. In terms of security level, proposed techniques offer protection against software-compromised MCUs including those that are physically re-programmed. However, physical invasive attacks (e.g., inducing hardware faults, snooping on the memory bus, or retrieving MCU secrets via physical side-channels) are out of scope. There are standard techniques that address such attacks [110].

This dissertation further discusses these and other specific limitations inline in the corresponding chapters.

1.3 Dissertation Structure

Next, Chapter 2 provides some background relevant to the dissertation as a whole, including its scope and required fundamental security services: Remote Attestation and Proof-of-Execution (*PoX*). Chapters 3 and 4 present, respectively, *Tiny-CFA* and *DIALED*, the runtime attestation architectures for detecting control-flow and data-only attacks in low-end MCUs. These two architectures are built atop *PoX* to support basic runtime attestation mechanisms. *DIALED* is directly built atop *Tiny-CFA* to further reduce hardware overhead. Chapter 5 focuses on sensor data privacy problem stated earlier. It defines sensor data privacy, *Privacy-from-Birth*, and presents a provable RoT – *VERSA* – that realizes *Privacy-from-Birth* with formally verified hardware. Next, Chapter 6 introduces *CASU*, a proactive RoT that prevents code modification attacks. *CASU* includes an efficient secure over-the-air update protocol suitable for low-end devices. Finally, Chapter 7 discusses directions for future work.

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 material. Section 2.1 sets the scope of targeted devices. We motivate this choice and discuss some intended contributions (specific contributions are outlined in subsequent chapters). Next, Section 2.2 overviews Remote Attestation ($\mathcal{RA}$): a security service that enables verification of the software state of a potentially compromised remote device – a prover ($\mathcal{P}\mathsf{rv}$) – by a trusted verifier ($\mathcal{V}\mathsf{rf}$). We also overview of *VRASED*, a formally verified $\mathcal{RA}$ architecture for low-end MCUs. Lastly, Section 2.3, describes Proof-of-Execution (*PoX*): an attestation service that provides proof that software executed properly, besides measuring its integrity. Following this, we overview *APEX*, a formally verified *PoX* architecture for low-end MCUs.

2.1 Targeted Devices

We focus on low-end CPS/IoT/smart devices with low computing power and meager resources. These are some of the smallest and weakest devices based on low-power single-core MCUs with only a few kilobytes (KB) of program and data memory. Two prominent examples of such MCUs are Atmel AVR ATmega [5] and TI MSP430 [74]: 8- and 16-bit CPUs, respectively, typically running at 1-16MHz clock frequencies, with $\approx$ 64KB of addressable memory.

Figure 2.1 illustrates a generic architecture of such MCUs. It includes a CPU core, a Direct Memory Access (DMA) controller, and an interrupt control logic connected to the main memory via a bus, on a single System-on-a-Chip (SoC). DMA is a hardware controller that can read/write from/to memory in parallel with the core. Main memory contains several logical regions: read-only memory (ROM), program memory (PMEM or flash), Interrupt Vector Table (IVT), data memory (DMEM or RAM), and peripheral memory. IVT stores pointers to the Interrupt Service Routines (ISRs), where the execution jumps when an interrupt occurs; it also contains the Reset Vector pointer from where the core starts to execute, after a reboot. Application software is installed in PMEM, which is non-volatile memory realized as flash, and it uses DMEM, which is volatile memory made up of DRAM, for its stack and heap. ROM contains the bootloader and/or any immutable software hard-coded at manufacturing time, which is not modifiable thereafter. Peripheral memory region (also in DRAM and often considered as a part of DMEM), contains memory-mapped I/O and other communication interfaces, i.e., addresses in the memory layout that are mapped to hardware components, e.g., timers, General-Purpose Input/Output (GPIO), Universal Asynchronous Receiver/Transmitter (UART), Analog-to-Digital Converter (ADC) and vice versa, Inter-Integrated Circuit (I2C), and Serial Peripheral Interface (SPI). In particular, GPIO is peripheral memory addresses hardwired to physical ports that interface with external circuits, e.g., analog sensors/circuits.

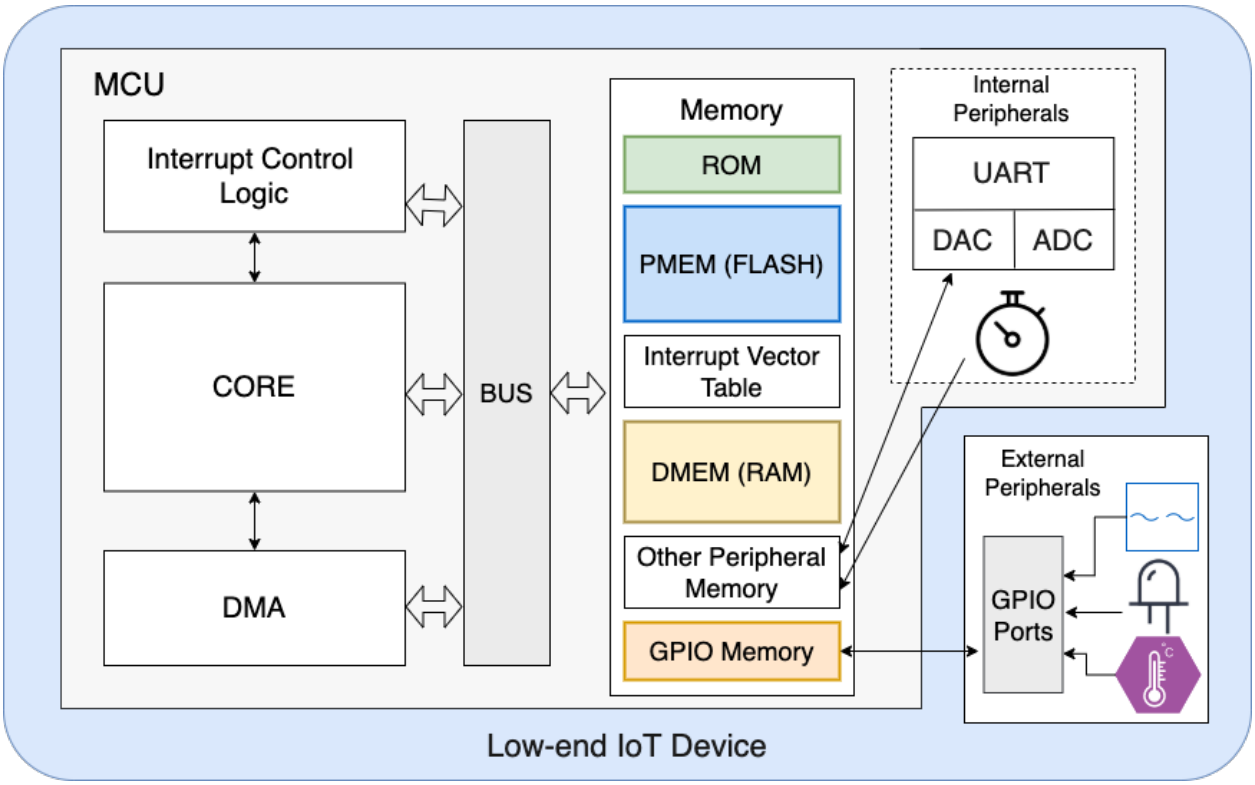

Figure 2.1: System Architecture of an MCU-based IoT Device

We note that small MCUs usually come in one of two memory architectures: Harvard and von Neumann. The former isolates PMEM and DMEM by maintaining two different buses and address spaces, while the latter keeps both PMEM and DMEM in the same address space and accessible via a single bus.

These MCUs execute instructions in place, i.e., directly from flash memory. They have neither memory management units (MMUs) to support virtualization/isolation, nor memory protection units (MPUs). Therefore, privilege levels and isolation used in higher-end devices and generic enclaved execution systems (e.g., Intel SGX [75] or ARM TrustZone-A [26]) are not applicable.

The prototype implementation of all the proposed RoTs in this dissertation is based on MSP430 MCU, a common platform for low-end embedded devices. One important factor in this choice is public availability of an open-source MSP430 MCU design – OpenMSP430

[68]. Nonetheless, we note that our generic machine model and methodology can be readily applicable to other low-end MCUs of the same class, such as Atmel AVR ATmega. Additionally, we hope that the lessons and insights gained from this work can be valuable for designing and proving the security of similar services targeting higher-end devices.

2.2 Remote Attestation ($\mathcal{RA}$)

Remote Attestation ($\mathcal{RA}$) allows a trusted entity (verifier = $\mathcal{V}$rf) to remotely measure current memory contents (e.g., software binaries) of an untrusted embedded device (prover = $\mathcal{P}$rv). As shown in Figure 2.2, $\mathcal{RA}$ is usually realized as a challenge-response protocol:

1. $\mathcal{V}$rf sends an attestation request containing a challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv.
2. $\mathcal{P}$rv receives the request and computes an authenticated integrity check over its memory and $\mathcal{C}$hal. The memory region can be either pre-defined or explicitly specified in the attestation request.
3. $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.
4. $\mathcal{V}$rf verifies the result and decides if it corresponds to a valid $\mathcal{P}$rv state.

The authenticated integrity check is accomplished by first measuring the $\mathcal{P}$rv software memory using a suitable cryptographic hash function (e.g., SHA-256). Then, the result, i.e. the digest, is authenticated using a Message Authentication Code (e.g., HMAC) or a digital signature (e.g., ECDSA). While computing a MAC requires $\mathcal{P}$rv to have a symmetric key shared with $\mathcal{V}$rf, computing a signature does not need any shared secrets, since $\mathcal{V}$rf uses $\mathcal{P}$rv's public key for verification. Both MAC and signature require secure storage to store the symmetric/private key that is not accessible to software running on $\mathcal{P}$rv memory, except for trusted and typically immutable attestation code (or attestation hardware engine, when

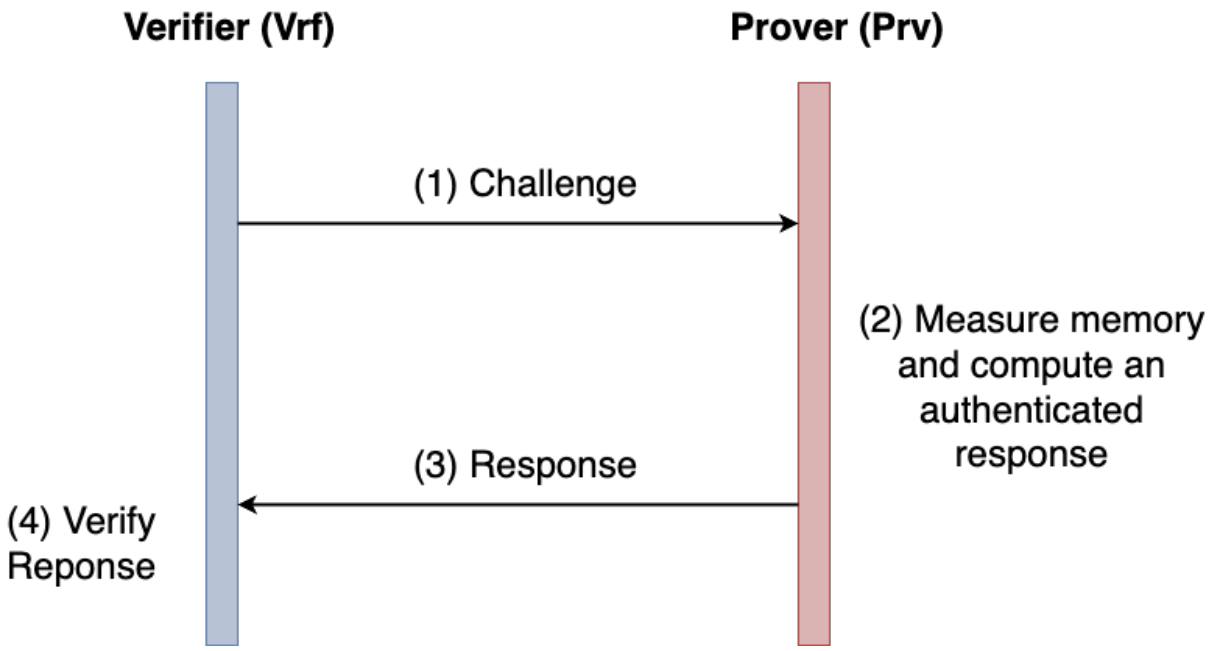

Figure 2.2: $\mathcal{RA}$ Protocol

present). Since we are dealing with low-end MCUs, we assume the symmetric key setting with MAC implementation, however, we note that using a public key with signatures is not very different. Since most $\mathcal{RA}$ threat models assume that $\mathcal{P}$rv's software is compromised, secure key storage implies some level of hardware support.

2.2.1 *VRASED*

VRASED [47] is a verified hybrid (hardware/software) $\mathcal{RA}$ architecture for for low-end MCUs. It comprises a set of (individually) verified hardware and software sub-modules; their composition provably satisfies formal definitions of $\mathcal{RA}$ soundness and security. *VRASED* software component, which is immutable (stored in ROM), implements the authenticated integrity function computed over a given "Attested Region" (AR) of $\mathcal{P}$rv's memory. Meanwhile, its hardware component assures that its software counterpart executes securely and that no function of the secret key is ever leaked. In short, $\mathcal{RA}$ soundness states that the integrity measurement must accurately reflect a snapshot of $\mathcal{P}$rv's memory in AR, disal-

lowing any modifications to AR during the actual measurement. $\mathcal{RA}$ security defines that the measurement must be unforgeable, implying protection of secret key $\mathcal{K}$ used for the measurement.

In order to prevent DoS attacks on $\mathcal{Prv}$, the RA protocol may involve authentication of the attestation request, before $\mathcal{Prv}$ performs attestation. If this feature is used, an authentication token must accompany every attestation request. (By saying "this feature is used", we mean that its usage, or lack thereof, is fixed at the granularity of a $\mathcal{Vrf}$-$\mathcal{Prv}$ setting, and not per single $\mathcal{RA}$ instance.). In *VRASED*, $\mathcal{Vrf}$ computes this token as an HMAC over $\mathcal{C}\mathsf{hal}$, using $\mathcal{K}$. Since $\mathcal{K}$ is only known to $\mathcal{Prv}$ and $\mathcal{Vrf}$, this token is unforgeable. To prevent replays, $\mathcal{C}\mathsf{hal}$ is a monotonically increasing counter, and the latest $\mathcal{C}\mathsf{hal}$ used to successfully authenticate $\mathcal{Vrf}$ is stored by $\mathcal{Prv}$ in persistent and protected memory. In each attestation request, incoming $\mathcal{C}\mathsf{hal}$ must be greater than the stored value. Once an attestation request is successfully authenticated, the stored value is updated accordingly.

VRASED software component is based on a formally verified HMAC implementation from the HACL* cryptographic library [139], which is used to compute:

$$H = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), AR) \tag{2.1}$$

where $KDF(\mathcal{K}, \mathcal{C}\mathsf{hal})$ is a one-time key derived from the received $\mathcal{C}\mathsf{hal}$ and $\mathcal{K}$ using a key derivation function.

All techniques proposed in this dissertation use *VRASED* for some form of attestation or HMAC computation on $\mathcal{Prv}$ memory. Details of each instance of *VRASED* usage are in each chapter.

2.3 Proofs of Execution (PoX)

PoX augments $\mathcal{RA}$ capability by proving to $\mathcal{Vrf}$ that: (a) the expected software is stored in $\mathcal{Prv}$ memory, and (b) this code executed, and any outputs were produced by its timely and authentic execution. In short, PoX cryptographically binds and authenticates the executed software and the outputs it generated during runtime, with the help of $\mathcal{RA}$.

2.3.1 $APEX$

The first PoX architecture for low-end MCU-s is in $APEX$ [49]. It includes a hardware module controlling the value of a 1-bit flag called $EXEC$, that cannot be written by any software. A value $EXEC = 1$ indicates to $\mathcal{Vrf}$ that attested code was executed successfully, between the time when the $\mathcal{Chal}$ was received from $\mathcal{Vrf}$ and the time when the $\mathcal{RA}$ measurement occurs (via HMAC). Similarly, when it receives an attestation reply with $EXEC = 0$, $\mathcal{Vrf}$ concludes that execution of said code did not occur, or the code terminated due to an error, or that execution (or its output) was tampered with. In $APEX$, the $\mathcal{RA}$ measurement covers: (a) the $EXEC$ flag itself; (b) the region where this execution's output is saved (output region – OR); and (c) the executable itself (stored in the executable region – ER). Thus, security of the underlying $\mathcal{RA}$ architecture guarantees that the contents of these memory regions cannot be forged/spoofed to something different from their values at the time of the attestation computation.

$APEX$ considers that a code executed properly (and sets $EXEC = 1$) if and only if:

1. Execution is atomic (i.e., uninterrupted), from the executable's first instruction (legal entry ER_{min}), to its last instruction (legal exit ER_{max});

2. Neither the executable (ER), nor its outputs OR are modified in between the execution

and subsequent $\mathcal{RA}$ computation;

3. During execution, data-memory (including OR) cannot be modified, by means other than the executable in ER itself, e.g., no modifications by other software or DMA controllers.

These conditions mean that $EXEC = 1$ assures that memory contents (of ER and OR) are consistent between ER's code execution and respective attestation, and that execution itself is untampered, e.g. via interrupts, or modification of intermediate results in data memory. ER and OR locations and sizes are configurable, allowing for *PoX* of arbitrary code and output sizes. *APEX* implementation is built atop *VRASED*. Moreover, *APEX* hardware module is itself formally verified to adhere to a set of formal logic specifications, similar to *VRASED*. These properties, along with *VRASED* verified guarantees, are proven sufficient to imply a security definition (stated using the cryptographic security game framework [92]) for unforgeable proofs of execution. For a detailed description of *APEX* proofs, we refer the interested reader to [49].

As discussed in [49], similar to Trusted Execution Environments (TEEs) targeting higher-end platforms (e.g., Intel SGX [75] and ARM TrustZone[26]), *APEX* assumes executable correctness, i.e., the code to be executed on $\mathcal{P}\mathsf{rv}$ is assumed to be bug-free and memory-safe code. Hence, *APEX* does not protect against runtime (aka control-flow and data-only) attacks. In *Tiny-CFA* and *DIALED*, we bridge this gap by introducing an automated code instrumentation technique that uses *APEX* to implement Control-Flow Attestation ($\mathcal{C}\mathsf{FA}$) and Data-Flow Attestation ($\mathcal{D}\mathsf{FA}$), respectively. In other words, we show that $\mathcal{C}\mathsf{FA}$ on top of *APEX* (or any *PoX*), without any additional hardware requirement, is both possible and affordable.

Chapter 3

Tiny-CFA: A Minimalistic Control-Flow Attestation Using Verified Proofs of Execution

Abstract

This chapter fills the gap in prior work by detecting control-flow attacks in low-end MCUs. Control-flow attacks refer to manipulation of software execution flow during runtime, which can lead to incorrect and potentially malicious behavior. While a few techniques have been proposed, they require a lot of additional hardware that might cost more than the underlying MCU.

To this end, we construct *Tiny-CFA*, a Control-Flow Attestation ($\mathcal{C}$FA) technique that requires only a single hardware capability: the ability to generate proofs of remote software execution (*PoX*). This results in the lowest hardware overhead among all $\mathcal{C}$FA techniques. Furthermore, in terms of runtime overhead, *Tiny-CFA* incurs lower performance compared to previous $\mathcal{C}$FA techniques. We implement and evaluate *Tiny-CFA*, analyze its security, and demonstrate its practicality using real-world publicly available applications.

Material in this chapter appeared in the Proceedings of 24th Design, Automation and Test in Europe Conference (DATE 2021) [53].

3.1 Introduction

Runtime/data-oriented attacks [125] tamper with execution state on the program's stack or heap to arbitrarily divert the program's execution flow. Such attacks need not modify the executable itself, but only the order in which its instructions are executed. Thus, they are not detectable by $\mathcal{R}$A. Control-flow 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 [45] can overwrite functions' return addresses, hijacking the program's control-flow and launching well-known Return-Oriented Programming (ROP) attacks [120]. 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 techniques, available in high-end platforms. We discuss a concrete example of such an attack in low-end MCU-s (and how it is detected by *Tiny-CFA*) in Section 3.4.1.

Control-Flow Attestation ($\mathcal{C}$FA) [15, 55, 54, 135] augments conventional $\mathcal{R}$A capability to enable detection of control-flow attacks. $\mathcal{C}$FA techniques provide $\mathcal{V}$rf with a report that conveys whether the expected code is loaded on $\mathcal{P}$rv, as well as which particular instruction path was taken during each execution of this program. In other words, $\mathcal{C}$FA provides 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. This is typically achieved by securely logging information associated with the destination of each control-flow altering instruction, e.g., `jumps`, `branches`, `returns`.

$\mathcal{C}$FA 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, hardware branch monitors, and hardware hash engines. However, for resource constrained MCU-s, these requirements are too costly, since their hardware overhead

is often higher than that of the MCU's core itself, in terms of size, energy, and monetary cost. To bridge this gap, our work uses *PoX* [49] (see Section 2.3 for details) – along with automatic code instrumentation, to derive a new $\mathcal{C}$FA technique. Since *PoX* can be implemented efficiently even on most resource-constrained MCU-s, our $\mathcal{C}$FA technique has considerably lower hardware overhead than that of prior work.

3.1.1 Contributions

This chapter makes the following contributions:

1. Design of *Tiny-CFA*– a $\mathcal{C}$FA technique based on automated software instrumentation where the only hardware requirement is that already provided (at relatively low-cost) by *PoX* architectures.

2. Implementation of *Tiny-CFA* based on *APEX* – as a result, its hardware cost is about 1 to 2 orders of magnitude lower than prior $\mathcal{C}$FA techniques. Hence, it is suitable for the low-end and ultra-low-energy MCU-s, such as MSP430 and AVR ATmega32. We also show a case study of a control-flow attack and how *Tiny-CFA* detects it.

We note that because *Tiny-CFA* implementation relies on a formally verified *PoX* architecture as the sole architectural component on $\mathcal{P}$rv, it is also the first $\mathcal{C}$FA technique to offer the high-level assurance provided by a verified hardware Trusted Computing Base (TCB).

3.2 Background

3.2.1 Control-Flow Attestation ($\mathcal{C}$FA)

Runtime attacks exploit program vulnerabilities to cause malicious and unauthorized program actions. The most prominent example is a buffer overflow, allowing the attacker to corrupt memory adjacent to a buffer. The main target of these attacks is the manipulation of control-flow information stored on the program's stack and heap, causing the program to deviate from its intended behavior. In addition to detection of code modification via $\mathcal{R}$A, $\mathcal{C}$FA detects such runtime attacks that hijack the program's control-flow by reporting the path the program took to $\mathcal{V}$rf.

A typical $\mathcal{C}$FA scheme works by first instrumenting the existing software to record every control-flow transfer it takes at runtime. After the program terminates, this log is then sent to the $\mathcal{V}$rf for investigation of potential control-flow attacks. The instrumentation ensures that at each control-flow altering instruction (e.g., `jumps`, `branches`, `returns`), execution is trapped into a secure region to log the destination address. Only logging each control-flow transfer is not sufficient, since one flavor of control-flow attacks involves hijacking the loops to iterate more or fewer times than expected. Therefore, $\mathcal{C}$FA also logs the condition on which such control-flow transfer occurs.

C-FLAT [15] is the earliest $\mathcal{C}$FA architecture. It uses ARM TrustZone-M [90] *secure world* to implement $\mathcal{C}$FA. By instrumenting the executable, at its every control-flow transfer, with context switches between TrustZone's normal and secure worlds, the control-flow path is logged into protected memory. C-FLAT targets higher-end embedded devices (e.g., Raspberry Pi), and its dependence on TrustZone (plus, numerous context switches) makes it unsuitable for low-end MCU-s.

3.3 *Tiny-CFA*

Tiny-CFA uses *APEXPoX* that ties the executed code to its output, stored in a data-memory range of configurable size, called OR. The basic idea is to instrument the code to produce a log of the program control-flow path and make it a part of the output. The program instrumentation writes the destination address of each control-flow altering instruction into OR. We denote this control-flow log as CF-Log.

As shown in Figure 3.1, both regular program outputs and CF-Log are written to OR. Recall from Section 2.3 that OR size/location is configurable. Hence, $\mathcal{V}\mathsf{rf}$ can choose OR to be large enough to fit both the regular program output and its expected CF-Log. Note that, in any $\mathcal{C}\mathsf{FA}$ scheme, $\mathcal{V}\mathsf{rf}$ must have *a priori* knowledge of the expected/benign control-flows and their sizes. Therefore, the appropriate OR size is trivially obtained by adding the regular output and CF-Log sizes. The regular program output is written to OR normally, bottom-to-top of OR, as in *APEX*. Whereas, *Tiny-CFA* instrumentation writes CF-Log to OR from top to bottom. This strategy is similar to how stack and heap are handled in RAM and it assures that the program output and CF-Log do not interfere or overlap with each other, as long as OR is appropriately sized.

We believe that this general idea is both intuitive and sensible; it guides *Tiny-CFA*'s design. However, ensuring that *Tiny-CFA* results in a *secure* $\mathcal{C}\mathsf{FA}$ design is more challenging. To see why, note that **the executable to be attested**, (i.e., security-critical code stored in ER) **is itself subject to control-flow attacks**. Thus, any values logged to CF-Log by the instrumented executable can, in principle, be modified as part of a control-flow attack. In other words, *Tiny-CFA*'s approach secure only **if** CF-Log is an **append-only log**. Otherwise, upon completion of its nefarious tasks, a control-flow attack can overwrite CF-Log to reflect a benign or expected control-flow, erasing any trace of the compromised control-flow and thus fool $\mathcal{V}\mathsf{rf}$. In higher-end $\mathcal{C}\mathsf{FA}$ architectures (e.g., C-FLAT [15]), this property is ob-

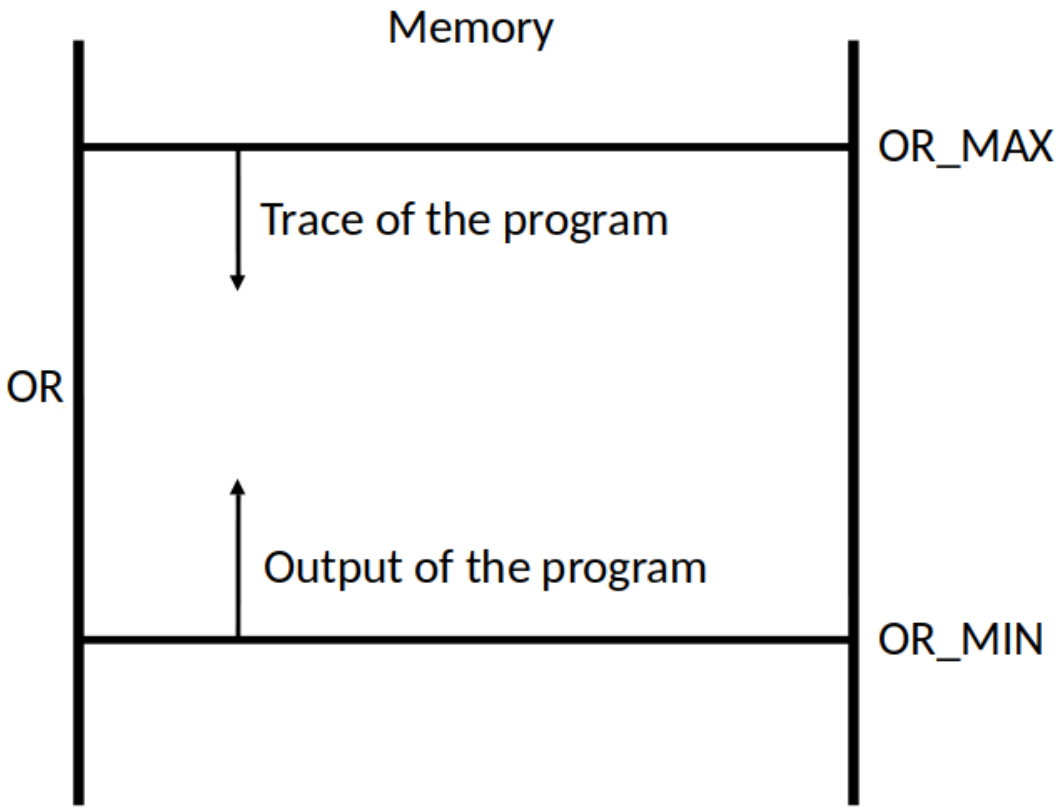

Figure 3.1: OR region used to store regular program outputs and CF-Log.

tained by logging the control-flow to dedicated secure memory, which is never accessible to untrusted/application code. However, as discussed in Sections 3.1 and 3.2, low-end MCU-s cannot afford such expensive security features. Below, we detail how *Tiny-CFA* can be made secure by relying exclusively on *PoX* and instrumentation.

3.3.1 Design Rationale & Security

We now discuss *Tiny-CFA* design rationale and security properties at a high-level. Implementation details of *Tiny-CFA* on MSP430 are specified in Section 3.3.3. We postulate the properties that ensure that control-flow attacks are always detected under the following comprehensive adversarial model:

3.3.1.1 Adversarial Model

We assume that the adversary controls $\mathcal{P}$rv's entire software state, including code and data. $\mathcal{A}$dv can modify any writable memory and read any memory that is not explicitly protected by hardware-enforced access control rules (e.g., *APEX* rules). Program memory modifications can be performed to change instructions, while data memory modifications may trigger control-flow attacks. Adversarial modifications are allowed before, during, or after the execution of the program.

3.3.1.2 Security Properties

The following security properties are adhered to by *Tiny-CFA* for achieving guaranteed $\mathcal{C}$FA.

(P1) Integrity of Code, Instrumentation, and Output

Clearly, any instrumentation-based approach is only sound if unauthorized modifications to the instrumented code itself (e.g., to remove instrumentation) are detectable. Detection of modifications is offered by the underlying $\mathcal{R}$A and *PoX* architectures (see Section 3.2). In particular, these architectures guarantee that any unauthorized code modification is detected by $\mathcal{V}$rf. They also guarantee that modifications to attested executable's output (*OR* – which includes CF-Log) are only possible if done by the attested executable itself, during its execution.

(P2) Secure logging of control-flow instructions

The first step in *Tiny-CFA*, is to instrument all control-flow altering instructions to log their destinations to CF-Log, in OR. CF-Log is implemented as a stack, from the highest value in

OR (OR_{max}) growing downwards, as shown in Figure 3.1. The pointer to the top of this stack is stored in a dedicated register $\mathcal{R}$. Each control-flow instruction is then instrumented with additional instructions to push its destination address to this stack, i.e.: (a) write the destination of address to the memory location pointed to by $\mathcal{R}$; and (b) decrement $\mathcal{R}$. At instrumentation time, the assembly code of the executable is inspected to assure that no other instructions utilize the MCU register $\mathcal{R}$. In all practical examples considered in this work, executables have at least one free register available. If no such register exists by default, the code can be recompiled to free up one register.

(P3) Secure logging of conditional branches

Conditional branches determine control-flow at runtime, depending on a result of a conditional statement, e.g., a comparison or equality test. These instructions are used to implement `loops` and `if-then-else` statements used in high-level languages. Conditional branches are instrumented by pushing to CF-Log's stack (using the same method as in **P2**) the possible destinations as well as the result of the conditional statement. This way, by inspecting CF-Log, $\mathcal{V}$rf can determine the exact path taken by the conditional branch.

(P4) CF-Log Write safety

Write operations (e.g. `mov`) are risky because they can be used to overwrite CF-Log, thus hiding the compromised control-flow from $\mathcal{V}$rf. Direct writes (which modify constant addresses) are easy to deal with because they can be statically inspected for safety at instrumentation time. In particular, the instrumenter can check that no direct writes modify CF-Log reserved addresses in OR. Indirect writes modify memory addresses determined at runtime. Consequently, they require instrumentation to check at runtime. After each indirect write, *Tiny-CFA* instrumentation inserts instructions to check whether the write destination

is within CF-Log by checking if the write destination is within the range $[\mathcal{R}, OR_{max}]$ – the memory range currently in use to store CF-Log. Upon detection of an illegal write, execution is terminated, implying an invalid control-flow.

(P5) Wrap-around attack protection

Another way for a control-flow attack to go undetected is to keep executing the program until $\mathcal{R}$ value overflows and wraps-around, and overwriting of CF-Log. To protect against such attacks, modifications to $\mathcal{R}$ have an additional check, ensuring that whenever $\mathcal{R}$ points to an instruction outside OR range, execution is terminated.

(P6) $\mathcal{R}$ initialization verification

Previous properties rely on $\mathcal{R}$, which is initialized to OR_{max} at the start of execution, to assure that CF-Log is indeed stored in OR. However, performing this initialization inside the executable being attested allows for control-flow attacks that jump back to the $\mathcal{R}$ initialization code to reset $\mathcal{R}$ in the middle of the execution. To prevent this, *Tiny-CFA* instruments the executable to check if $\mathcal{R}$ has been previously properly initialized to $\mathcal{R} = OR_{max}$. The caller application becomes responsible for initializing $\mathcal{R}$ appropriately, making control-flow attacks that re-initialize $\mathcal{R}$ to reset CF-Log impossible, since they require jumping outside of the executable range – ER – which is detected by *APEX* as a violation.

3.3.1.3 Security Argument

Below we argue the security of *Tiny-CFA* based on the aforementioned properties (**P1 - P6**).

Let $\mathcal{P}$ denote a function/code-segment for which execution and control-flow need to be attested. Properties **P2 & P3** assure that all changes to the control-flow of $\mathcal{P}$ are logged to CF-Log at runtime. Then, by inspecting an authentic (untampered) CF-Log, $\mathcal{V}\mathsf{rf}$ can determine the exact control-flow taken by that particular $\mathcal{P}$ execution. Meanwhile, properties **P5 & P6** guarantee that CF-Log is stored inside OR, within $[\mathcal{R}, OR_{max}]$ range. Property **P4** detects any illegal writes during execution that attempt to modify CF-Log, i.e., writes to $[\mathcal{R}, OR_{max}]$ range. Hence, for a given execution of $\mathcal{P}$, the combination of P4, P5 & P6 guarantees that each written value can never be overwritten or deleted from CF-Log. Finally, **P1**, inherited from the underlying *PoX* architecture, assures that neither $\mathcal{P}$ instructions (including instrumentation), nor its output (including CF-Log) can be modified by other means (e.g., other software on $\mathcal{P}\mathsf{rv}$, interrupts, DMA) before, during, or after execution. Any such attempt is detectable by $\mathcal{V}\mathsf{rf}$, because it causes *APEX* to set $EXEC = 0$; recall the $EXEC$ flag behavior described in Section 2.3. Therefore, *Tiny-CFA* properties **P1-P6** suffice to implement secure $\mathcal{C}\mathsf{FA}$, under the aforementioned adversarial model. □

3.3.2 Optimizations

CF-Log size determines the practicality of *Tiny-CFA* due to the resource-constrained nature of low-end MCU-s, especially, with respect to memory size. Any log of control flow transitions bloats rapidly for control-flow intensive code segments, e.g., loops with many iterations. In this section, we discuss two simple optimizations **(O1 & O2)** that significantly reduce CF-Log size without sacrificing overall security.

(O1) Static Control-Flow Instructions

Control-flow instructions with constant destination addresses (determined statically in the code) need not be logged to CF-Log, as their effect on the program control-flow can not

change at runtime. This removes the need to log operations, such as usual function calls (with exception of callbacks), fixed-address `go-to-s`, and similar.

(O1) Loops Optimization

Loops are challenging to log efficiently due to their high number of control-flow operations. For instance, consider a delay function based on *busy-wait*, commonly used in MCU code. It essentially consists of a loop that increments a counter up to a certain constant. The higher the delay, the higher the number of iterations, implying the higher the number of control-flow instructions to be logged. Even a 1-second delay loop, would require millions of iterations (assuming typical clock frequencies on the order of MHz) resulting in millions of symbols logged to CF-Log. To deal with such cases, we introduce an optimization for loops for which the number of iterations can be determined, at instrumentation time.

Tiny-CFA instrumenter inspects each conditional branch. For each loop branch instruction bi, changing the control-flow to destination instruction di, the instrumenter inspects all instructions in the range $[bi, di]$. If $[bi, di]$ contains no indirect control-flow instructions (e.g., unconditional branches) and no nested branches (such as loop in loop, or if-else in loop), then the number of iterations caused by such a loop can be determined exclusively by the loop condition and the variables involved in this condition. For example, consider a delay loop `while (i < j) i++;`. The instrumenter tracks `i` and `j` based on their increment logic. In this case, since `i` increments by 1, it is `j - i`. The instrumenter logs the value of `j - i` along with the increment constant `1`. Therefore, instead of logging each branch at every iteration, *Tiny-CFA* simply logs the condition and the increment itself, only once. This allows $\mathcal{V}$rf to learn the exact control-flow generated by a loop (i.e., # iterations) without bloating CF-Log. In our 1-second delay example, instead of logging millions of symbols, the loop would log just a couple of bytes, corresponding to the loop exit condition. This optimization also applies to loops used in common memory/array manipulations, e.g., in

`memset`, and `memcpy`.

3.3.3 Implementing *Tiny-CFA*

We now describe how properties **P1-P6** are securely implemented via automatic assembly instrumentation on the MSP430 MCU. Our instrumenter is implemented in `Python` with approximately 300 lines of code. It replaces relevant code segments in order to securely enforce properties **P1-P6**.

```
1
2
3
4
5    ret
```

(a) Original

```
1    mov r1, @r4
2    dec r4
3    cmp #OR_MIN, r4
4    jn .L11
5    ret
```

(b) Instrumented

Figure 3.2: Instrumentation example: indirect control-flow instructions.

Figure 3.2 shows the instrumentation of indirect control-flow instructions: *return* in this particular example. It writes the return address, which in MSP430 assembly must be loaded to register $r1$ before `ret` is called, to CF-Log. In our implementation $\mathcal{R} = r4$. Hence, the content of $r1$ (destination address) is copied to the address pointed to by $\mathcal{R}$ in OR, as required by **P2**. To also enforce **P5**, upon writing to the address of $\mathcal{R}$, and moving $\mathcal{R}$ to point to the next address, the comparison at line 3 checks if $\mathcal{R}$ is still inside OR, otherwise exiting the program, by jumping to an exit instruction at line 4.

Figure 3.3 depicts the instrumentation of indirect write instructions to enforce **P4**. Upon writing to a given memory location (address pointed to by $r14$, in this example), checks are performed to determine if this write operation did not modify CF-Log memory range: $[\mathcal{R}, OR_{max}]$. If an illegal write occurs, program execution is terminated (at line 5) and a control-flow attack attempt is detected.

```
1   mov.b r15, @r14
2
3
4
5
6
7   ...
```

(a) Original

```
1     mov.b r15, @r14
2     cmp r4, r14
3     jlo .L12
4     cmp #OR_MAX, r14
5     jlo .L11
6   .L12:
7   ...
```

(b) Instrumented

Figure 3.3: Instrumentation example: indirect write instructions.

```
1   application:
2
3
4   ...
```

(a) Original

```
1   application:
2     cmp #OR_MAX, r4
3     jne .L11
4   ...
```

(b) Instrumented

Figure 3.4: Instrumentation example: $\mathcal{R}$ initialization check.

Figure 3.4 shows the instrumentation, required by **P6**, at the beginning of the code segment. It ensures that $\mathcal{R}$ is properly initialized, otherwise halting execution at line 3.

Finally, Figure 3.5 depicts the instrumentation required by **P3**. It logs to CF-Log the results of conditional statements. Note that, after a conditional statement (e.g., at line 1) evaluation, the result is stored in the status register $r2$. Hence, the content of $r2$ is written to CF-Log (line 2), since it is sufficient to determine the destination of the conditional branch. The

```
1     cmp.b #64, r15
2
3
4
5
6
7     jne .L2
8   ...
```

(a) Original

```
1     cmp.b #64, r15
2     mov r2, @r4
3     dec r4
4     cmp #OR_MIN r4
5     jn .L11
6     mov l(r4), r2
7     jne .L2
8   ...
```

(b) Instrumented

Figure 3.5: Instrumentation example: conditional branches.

same check to enforce **P5** in Figure 3.2, is also performed in this case, because information is being written to CF-Log. Since this check itself overwrites $r2$, the original value of $r2$ needs to be retrieved (at line 6) before the actual branch instruction at line 7. A more memory-optimized implementation of **P3** could be to rather find out which status bit in the status register decides the jump to the destination address and write only that bit into OR instead of the whole 16-bit register value. Essentially saving 15-bits of the CF-log for every conditional jump.

Remark: *Tiny-CFA can not be abused by control-flow attacks that jump in the middle of the instrumentation instructions. Such an illegal jump is logged to CF-Log and is thus detectable by* $\mathcal{V}$rf. *Since* $\mathcal{R}$ *never retracts (within a given execution), write checks (see Figure 3.3) make it impossible to delete any information logged to CF-Log, including jumps into the middle of instrumented code instructions.*

3.4 Case Study & Evaluation

3.4.1 Case Study: Control-Flow Attacks in Low-End MCU-s

Control-flow attacks can be extremely harmful, especially, for low-end devices used for safety-critical tasks. To illustrate this point, we show an attack on a medical syringe pump application implemented on a low-end MCU. For clarity, we focus on a simplified version of the OpenSyringePump application (available at: `https://github.com/naroom/OpenSyringePump`). Later, in Section 3.4, we evaluate *Tiny-CFA* on three applications, including the original OpenSyringePump code, which is longer and more complex than the example used here. OpenSyringePump was also used to motivate and evaluate prior $\mathcal{C}$FA approaches, e.g., C-FLAT.

```
1   int dose = 0;
2
3   void injectMedicine(){
4     if (dose < 10){  //safety check preventing overdose
5       P3OUT = 0X1;
6       delay(dose*time_per_dose_unit);
7     }
8     P3OUT = 0x0;
9   }
10
11  void parseCommands(int *recv_commands, int lenght){
12    int copy_of_commands[5];
13    memcpy(copy_of_commands, recv_commands, lenght);
14    dose = processCommands(copy_of_commands);
15    return;
16  }
```

Figure 3.6: Safety critical application exploitable by control-flow attacks.

Consider the `C` code segment in Figure 3.6. In this application, the MCU is connected through the general-purpose input/output (GPIO) port *P3OUT* (used at lines 5 and 8) to an actuator, responsible for injecting a given dose of medicine, determined in software, according to commands received through the network, e.g., from a remote physician. The function `injectMedicine` injects the appropriate dosage according to the variable `dose`, by triggering actuation for an amount of time corresponding to the value stored in `dose`. To guarantee a safe dosage, the `if` statement (at line 4) assures that the maximum injected dosage is 9, thus preventing overdosing.

Dosage is determined according to a list of values, e.g., symptom severity measures received from a remote physician. The function `parseCommands` (line 11) is responsible for making a copy of the received values and processing them to determine the appropriate dosage. However, this function can also be used to trigger a buffer overflow attack, leading to a malicious control-flow path. Specifically, because the size of `copy_of_commands` is static and equal to 5, an input array of larger size can cause other values in the program's stack to be overwritten, beyond the area allocated for `copy_of_commands`, and including the memory location storing the return address of `parseCommands`. In particular, the return

address is overwritten with the value of `recv_commands[5]`. By setting the content of `parseCommands[5]` to the address of line 5 in Figure 3.6, such an attack causes the control-flow to jump directly to line 5 (when `parseCommands` returns), skipping the safety check at line 4, and potentially overdosing the patient.

The above attack example is detectable neither by static $\mathcal{R}$A techniques nor by *PoX* techniques, since expected (unmodified) code still executes in its entirety, yet in an unexpected order. *Tiny-CFA*, however, detects such control-flow attacks, because the instrumentation of indirect control-flow instructions (e.g., `return` in Figure 3.2) commits the maliciously overwritten return address to CF-Log.

In Section 3.4 we evaluate *Tiny-CFA* performance in 3 realistic safety-critical applications: (1) OpenSyringePump – the full implementation of our toy example in Figure 3.6; (2) FireSensor (available at: `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor`) – a fire detector based on temperature and humidity sensors; and (3) UltrasonicRanger (available at: `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger`) – a sensor used by parking assistants for obstacle proximity measurement.

3.4.2 Experimental Results

Recall that, since *Tiny-CFA* requires no hardware support beyond that already provided by *APEX* [49], its hardware costs remain consistent with *APEX*. As a clear advantage over prior techniques, our approach requires 5.4 times fewer additional LUTs and 50 times fewer additional registers than the second cheapest approach – LiteHAX; see comparison of *Tiny-CFA* hardware overhead with other $\mathcal{C}$FA techniques in Figures 3.7 and 3.8. Hardware costs are as reported in the original papers [135, 54, 55, 49]

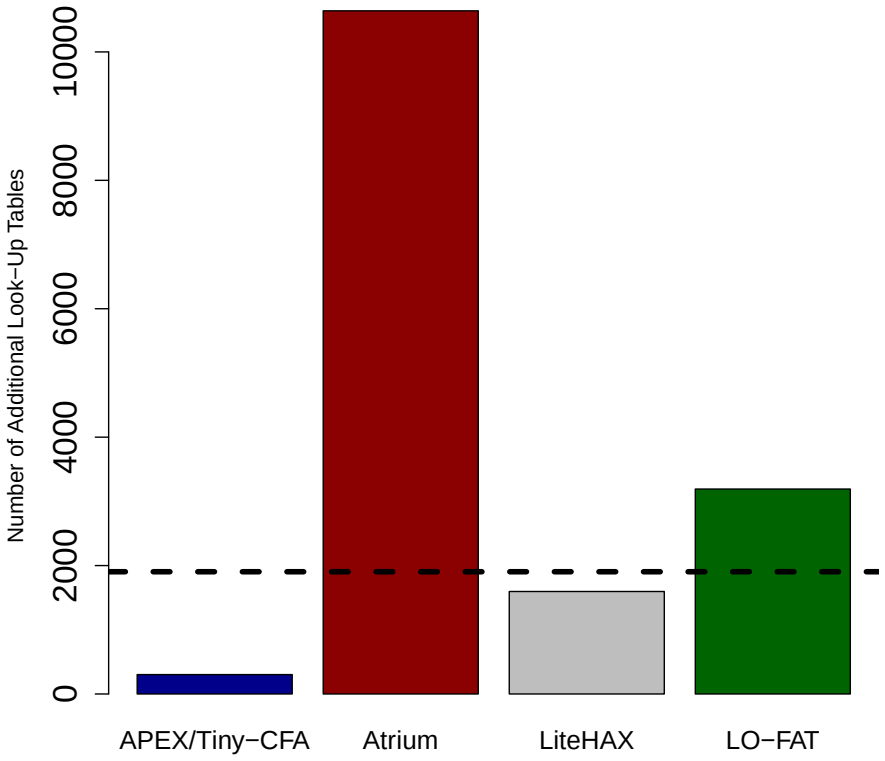

Figure 3.7: *Tiny-CFA* Additional HW overhead (%) in Number of Look-Up Tables. Dashed lines represent the total hardware cost of MSP430 core itself.

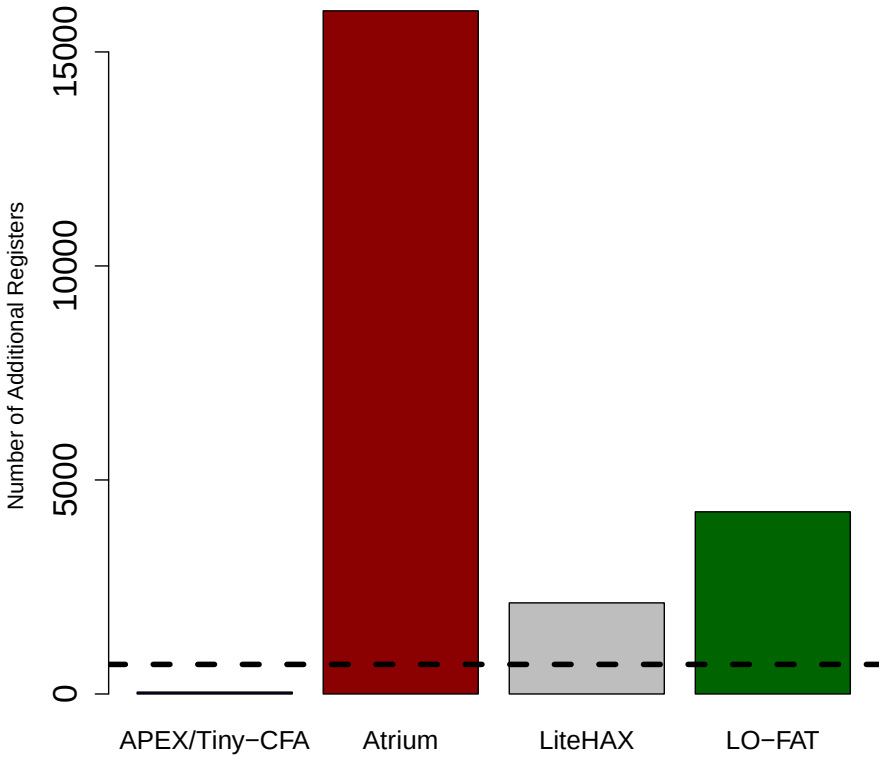

Figure 3.8: *Tiny-CFA* Additional HW overhead (%) in Number of Registers. Dashed lines represent the total hardware cost of MSP430 core itself.

3.4.2.1 Instrumentation Overhead

We measure the instrumentation overhead of *Tiny-CFA* based on three metrics: code size increase, runtime overhead, and CF-Log size. As mentioned in Section 3.4.1, our evaluation instantiates *Tiny-CFA* on MSP430 with three real-world, publicly available, and safety-critical use cases: `SyringePump`, `FireSensor`, and `UltrasonicRanger`. Tables 3.1 and 3.2 present experimental results for these three applications in their unmodified forms and when instrumented by *Tiny-CFA*. In each case, the attested execution corresponds to one iteration of the application's main loop (i.e., the application can report to $\mathcal{V}$rf with the attestation response once per iteration), involving the respective sensing and actuation tasks.

Table 3.1: Original application costs

	SyringePump	FireSensor	UltrasonicRanger
Code Size	218 bytes	434 bytes	238 bytes
Runtime	159644 cycles	20919 cycles	2799 cycles

Table 3.2: Instrumented application costs

	SyringePump	FireSensor	UltrasonicRanger
Code Size	416 bytes	790 bytes	442 bytes
Runtime	162218 cycles	31818 cycles	3027 cycles
CF-Log size	400 bytes	2068 bytes	30 bytes

In all three cases, code size increases by $\approx 80\%$, while CF-Log size ranges between 30 and $2k$ Bytes, and runtime overhead varies between $\approx 2\%$ and $\approx 50\%$. CF-Log size depends on the number of control-flow transfers occurring in the application. Programs performing simple tasks need smaller log size ($< 1k$ bytes), while those with complex tasks would need larger log sizes.

Tiny-CFA exhibits lower runtime overhead than C-FLAT [15]. C-FLAT is only evaluated using the `SyringePump` example, and its reported runtime overhead is $\approx 76\%$, due to instrumentation of trampolines and context switches; see [15] for details. Meanwhile, in all considered applications, *Tiny-CFA* runtime overhead remains below $\approx 50\%$. This is justified

by: (1) simpler design that does not rely on trampoline hypercalls or context switches, and (2) optimization **O2**, which removes per-iteration instrumentation away from delay loops. Since delay loops are used frequently in sensing/actuation applications, this optimization comes in handy in most practical scenarios. However, we do not compare runtime overhead of *Tiny-CFA* with Lo-FAT and LiteHAX since these two techniques do not instrument code, instead detecting branches in hardware.

In summary, experimental results indicate that, in all sample applications, instrumented executables remain well within the capabilities of low-end MCU-s, thus supporting *Tiny-CFA*'s practicality.

3.5 Conclusions

In this chapter, we designed, implemented, and evaluated *Tiny-CFA*: a low-cost $\mathcal{C}$FA approach targeting low-end MCU-s. *Tiny-CFA* couples a formally verified *PoX* architecture with automated code instrumentation to yield an effective low-cost $\mathcal{C}$FA. We argued security of *Tiny-CFA* and demonstrated, via a MSP430-based implementation, its ability to detect control-flow attacks.

Chapter 4

DIALED: Data Integrity Attestation for Low-end Embedded Devices

Abstract

Although $\mathcal{C}$FA techniques detect control-flow attacks, they cannot detect a more subtle class of data-only attacks. These attacks manipulate the software's execution without modifying the program code or its control flow. Instead, they exploit vulnerabilities in the code to corrupt intermediate computation results stored in the data memory, leading to unexpected and potentially harmful outcomes. In this chapter, we propose, implement and evaluate *DIALED*, the first Data-Flow Attestation ($\mathcal{D}$FA) technique applicable to the low-end MCUs. *DIALED* works in tandem with *Tiny-CFA* to detect all (currently known) types of runtime software exploits at a fairly low cost.

Research presented in this chapter appeared in the Proceedings of the 58th Design Automation Conference (DAC 2021) [52].

4.1 Introduction

Similar to control-flow attacks, data-only attacks exploit vulnerabilities in benign code to corrupt intermediate values in data-memory. It is well-known [125, 77] that data-only attacks need not alter the code or its control-flow in order to corrupt data. Without a way to detect data-only attacks (as well as code and control-flow modifications) results of $\mathcal{P}$rv's remote computation cannot be trusted. Hence, besides $\mathcal{R}$A and $\mathcal{C}$FA, there is also a need for attesting the data-flow of the program – Data-Flow Attestation ($\mathcal{D}$FA) – for providing a comprehensive report of what happened on $\mathcal{P}$rv during runtime.

Very recently, a technique supporting both $\mathcal{C}$FA and $\mathcal{D}$FA, called OAT [124], was proposed. However, it implements $\mathcal{D}$FA by relying on trusted hardware support from ARM TrustZone, which is only available on higher-end platforms (e.g., smartphones, and Raspberry Pi) and is not affordable to low-end, low-energy MCUs. In addition, OAT's security relies on the application programmer's ability to correctly annotate all critical variables in the code to be attested. This is a strong assumption since most control-flow and data-only exploits are caused by implementation bugs introduced by the very same application programmer. Naturally, it would be beneficial for this assumption to be avoided.

To address these limitations, we propose a new $\mathcal{D}$FA technique that leverages *PoX* and $\mathcal{C}$FA in conjunction with code instrumentation, similar to the approach used in Tiny-CFA. This approach aims to provide a solution that is applicable to low-end MCUs, avoiding the reliance on expensive trusted hardware support and mitigating the assumption of correct variable annotation by the application programmer. By combining *PoX*, $\mathcal{C}$FA, and code instrumentation, our proposed technique offers a comprehensive solution for both control-flow and data-only attacks, while remaining suitable for resource-constrained embedded devices.

4.1.1 Contributions

This chapter makes the following contributions:

1. Design of *DIALED*: Data Integrity in Attestation for Low-end Embedded Devices – a $\mathcal{D}$FA technique based on automated code instrumentation using *PoX* as its sole hardware requirement.

2. Implementation of *DIALED* based on *Tiny-CFA* ($\mathcal{C}$FA) and *APEX* (*PoX*). This reduces the hardware overhead significantly compared to prior architectures (such as OAT) making it suitable for resource-constrained MCU-s.

DIALED uses *APEX* to securely log and authenticate any data inputs used by the program. This authenticated log allows $\mathcal{V}$rf to reconstruct the entire data-flow of the program's execution, thus enabling detection of any data-corruption attacks via abstract execution of the attested program.

In the rest of this chapter, we describe *DIALED*'s design and analyze its security. We also report on the implementation of *DIALED* along with Tiny-CFA on TI MSP430 MCU and demonstrate its cost-effectiveness in the context of for three applications.

4.2 Background

This section overviews data-only attacks; it also compares such attacks with control-flow attacks and elaborates on how they cannot be detected by $\mathcal{C}$FA alone.

4.2.1 Control-Flow vs. Data-Only Attacks

Both control-flow and data-only attacks violate program execution integrity without modifying the actual executable, by taking advantage of implementation bugs, e.g., lack of array bound checks. Such vulnerabilities are quite common in memory-unsafe languages, such as C, C++, and Assembly, which are widely used to program MCUs.

Control-flow attacks change the order of instructions execution thus changing program behavior, escalating privilege, and/or bypassing safety checks (as shown in Figure 3.6). Whereas, data-only attacks only corrupt the critical data variables to produce wrong results without changing the order of execution.

```
1   int set = 0x1; // configured to cause actuation on Port 1
2   int settings[8]; // default settings produce dose = 5ul
3
4   void injectMedicinePort1(int new_setting, int index){
5     settings[index] = new_setting;
6     int dose = defineDosage(settings);
7     if (dose < 10){  //safety check preventing overdose
8       P3OUT = set;
9       delay(dose*time_per_dose_unit);
10    }
11    P3OUT = 0x0;
12    return;
13  }
```

Figure 4.1: Embedded application vulnerable to a data-flow attack.

As mentioned earlier, $\mathcal{C}$FA securely logs all control-flow transitions, enabling detection of the control-flow attack. However, $\mathcal{C}$FA cannot detect data-only attacks that do not change the control-flow. Figure 4.1 presents an implementation vulnerable to data-only attacks. To see this vulnerability, note that P3OUT register controls multiple physical ports, each associated with one bit:

- Setting P3OUT = 00000000 = 0x0 turns all physical ports off

- Setting P3OUT = 00000001 = 0x1 turns on 1st physical port
- Setting P3OUT = 00000010 = 0x2 turns on 2nd physical port
- Setting P3OUT = 00000011 = 0x3 turns on both 1st and 2nd physical ports, etc.

In order to trigger actuation through the proper port (Port 1 in this example), this code needs to set P3OUT = 0x1. This is configured in the global variable `set`, at line 1 of Figure 4.1. The value of `set` is later used to trigger actuation at line 8. The code also allows `settings` to be updated at an arbitrary position defined by the input parameter `index`. Since `settings` has a fixed length of 8, a malicious input with `index` = 8 would overflow this buffer causing `set` to be overwritten with the value of `new_setting`. An input `new_setting=0` with `index` = 9 would overwrite `set` = 0. Later, in line 8, when *set* is used to trigger actuation of port 1, it will instead have no actuation effect (since it is now 0). Consequently, the medicine will not be injected. It is important to note that this attack does not change the program control flow, but just corrupts data. It therefore can not be detected by $\mathcal{C}$FA alone.

In this chapter, we address detection of these data-only attacks by proposing a $\mathcal{D}$FA architecture – *DIALED*. Its core idea is to log (and send to $\mathcal{V}$rf) all inputs during execution along with its control flow, which enables $\mathcal{V}$rf to emulate the entire execution of the attested program, allowing it to detect both control-flow and data-flow attacks. *DIALED* is detailed in the next section.

4.3 *DIALED* Design

Figure 4.2 shows the components of *DIALED*: it is implemented alongside *Tiny-CFA* (itself based on *APEX*) to provide both $\mathcal{C}$FA and $\mathcal{D}$FA. The executable is separately instrumented by both *Tiny-CFA* and *DIALED*. *APEX* provides a proof of the execution of the *instrumented*

executable, serving as an authenticator for its output: a log containing the executable's control flow and its data inputs.

4.3.1 Overview

DIALED uses a novel input detection method via secure instrumentation of the executable. This instrumentation guarantees that all relevant data is logged during program execution, this is in addition to the control-flow log produced by *Tiny-CFA*. The underlying *PoX* provides $\mathcal{V}\mathsf{rf}$ with a proof that this output was indeed produced by the execution of the expected (instrumented) code. In doing so, *DIALED* provides $\mathcal{V}\mathsf{rf}$ with all information needed to execute this program locally and detect any code, control flow, or data compromises. The core property of *DIALED* is detection and secure logging of every external input received during program execution, including from peripherals, the network, and GPIO, as well as data fetches from memory locations outside the executable's own state (i.e., stack and heap).

Similar to OAT [124], the goal is to attest **embedded operations**, i.e., finite and self-contained safety-critical functions called by the program's main loop. Examples include sensing and actuation tasks triggered by commands received through the network, as in Section 4.2.1. Since embedded operations typically have well defined and reasonably small number of data inputs, *DIALED* can efficiently save all inputs to an append-only log – Input Log (I-Log). *DIALED* instrumentation assures that all *data inputs* are appended to I-Log. We define data inputs as follows:

Definition 4.1 (Data Inputs)**.** *Any value read from any memory location* ***outside*** *of the attested program's current stack. The program's current stack is the region located within the current stack pointer value (top of the stack) and the value of the stack pointer when the attested program was first called (based of the program's stack). It includes all local variables.*

According to this definition, read instructions that move/copy data from peripherals, network

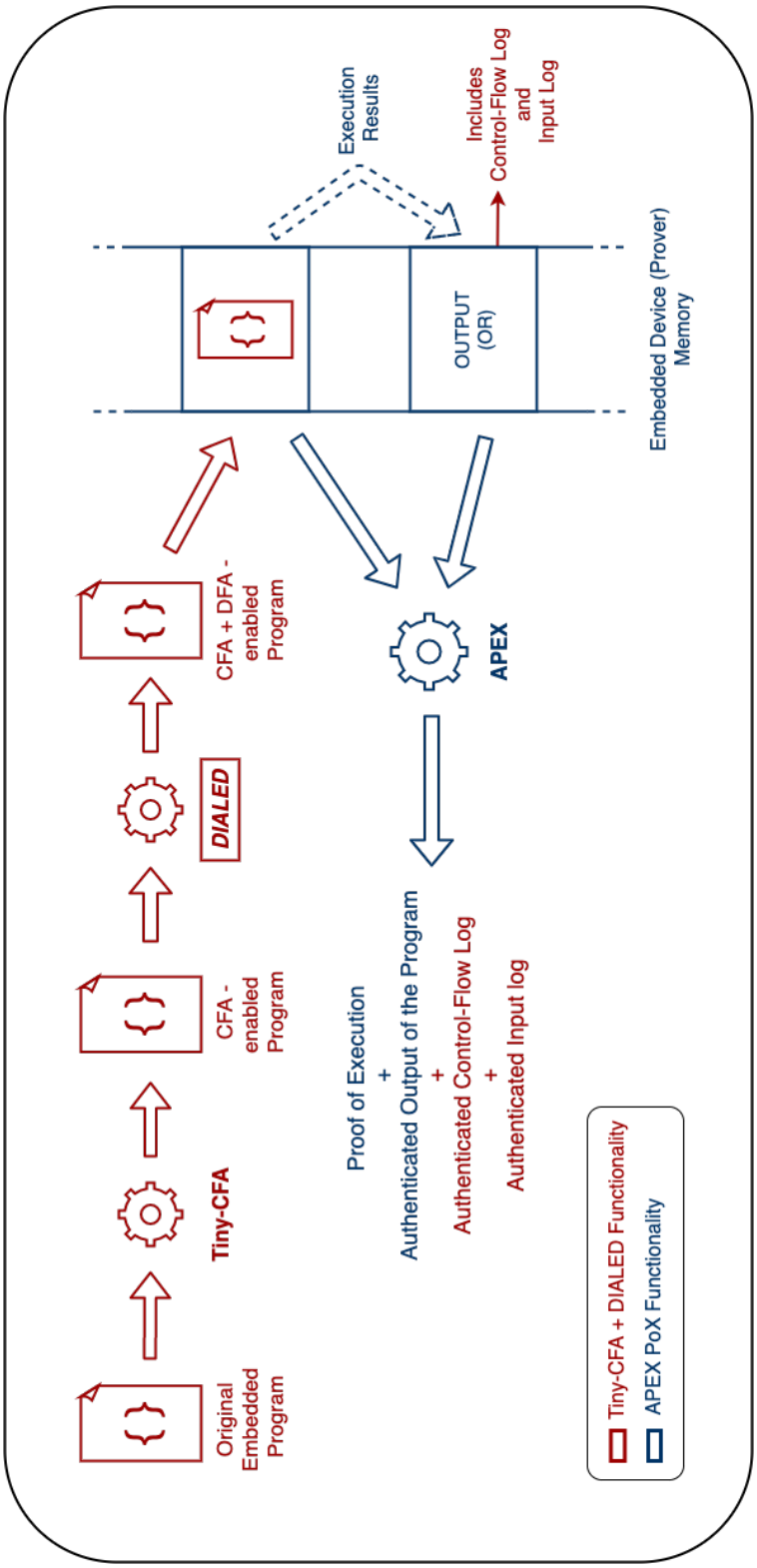

Figure 4.2: *DIALED* Architectural Components.

or GPIO are considered as Data Inputs and written to I-Log, since these involve reads from memory outside of the program's stack. But reads that occur during regular computation, e.g., instructions that compute on local variables are not written to I-Log, as they are not inputs. This approach makes the size of I-Log relatively small, which is confirmed by the real-world embedded operations considered in our evaluation in Section 4.5.

Recall that *Tiny-CFA* instruments the executable to produce a Control-Flow Log (CF-Log). In *DIALED*, both CF-Log and I-Log are written to *APEX*-designated output region *OR*. Hence, $\mathcal{V}$rf is assured of the integrity of these logs. In addition, given the attestation guarantee, $\mathcal{V}$rf is also assured that the correct/expected instrumented code was executed to produce this log. By knowing the code, its control-flow, and all inputs, $\mathcal{V}$rf can locally emulate its execution and verify all steps in this computation, as well as detect all data-only and control-flow attacks.

In the rest of this section, we discuss *DIALED*'s properties and its security. Then, Section 4.4 details *DIALED*'s instrumentation and architectural components.

4.3.2 Adversary Model

We assume an adversary that controls $\mathcal{P}$rv's entire software state, including code and data. It can modify any writable memory and read any memory that is not explicitly protected by hardware-enforced access controls, e.g., *APEX* rules. Program memory modifications can change instructions, while data memory modifications can trigger control-flow and data-only attacks arbitrarily. Adversarial modification attempts are allowed before, during, or after the execution.

4.3.3 Design Rationale

DIALED's security is based on five properties: **P1-P5**. We describe them at a high level in this section and discuss how *DIALED* achieves them. Subsequently, in Section 4.4, we describe implementation details of *DIALED* on MSP430 through automated code instrumentation.

(P1) Integrity Proofs for Code, Instrumentation, and Output

As an instrumentation-based technique, *DIALED* is only secure if any modifications to the instrumented code itself (e.g., removing instrumented instructions) is detectable. Detection of code modifications is already offered by the underlying *APEX PoX* architecture (see Section 2.3). *APEX* guarantees that every code modification is detected by $\mathcal{V}\mathsf{rf}$. It also guarantees that any modification of the attested executable's output region *OR* (which, in our case, includes CF-Log and I-Log) can only be done by the attested executable itself, during its execution.

(P2) Integrity Proof for the Control Flow

Since *DIALED* relies on instrumented instructions, these instructions can not be skipped, e.g., via control-flow violations. Therefore, *Tiny-CFA* ensures that the control flow is logged to CF-Log and whatever is written to CF-Log can not be modified; see Section 3 for details. Hence, all attempts to skip the logging of any data inputs are detectable by $\mathcal{V}\mathsf{rf}$ using CF-Log. The integrity of CF-Log itself is important to *DIALED*'s overall functionality since $\mathcal{V}\mathsf{rf}$ needs both CF-Log and I-Log in order to abstractly execute the program and verify the integrity of the execution.

(P3) Secure Logging of Data Inputs from Operation Arguments

To enable re-execution by $\mathcal{V}$rf, any arguments passed to the program at invocation must be securely logged to I-Log. *DIALED* automatically instruments the executable with Assembly instructions that copy all program arguments to I-Log.

(P4) Secure Logging of Runtime Data Inputs

Data inputs can be obtained at runtime, e.g., values read from GPIO, or packets arriving from the network. Such inputs are received through peripheral memory, at a particular set of physical addresses in data-memory. *DIALED* instruments every `read` instruction to check whether the read address is outside the program's stack. The range of the stack is determined by $[ls, hs]$, where ls is the value of the stack pointer saved at the moment when execution starts (before the allocation of local variables), and hs always reflects the current stack pointer, i.e., the top of the stack.

(P5) I-Log and CF-Log Integrity

To ensure integrity of CF-Log and I-Log, *DIALED* must guarantee that control flow and data-only attacks do not overwrite these logs. Thus, we realize I-Log and CF-Log as a single stack data structure inside OR, from the highest value (OR_{max}) growing downwards. The pointer to the top of this stack is stored in a dedicated register $\mathcal{R}$. Each instruction that alters the control flow or involves data input is instrumented (with additional instructions) to push the relevant values (either control-flow destination or data input) onto the stack, i.e.:

1. Write the value (destination of address or data input) to the location pointed by $\mathcal{R}$; and
2. Decrement $\mathcal{R}$.

At instrumentation time, assembly code is inspected to ensure that no other instructions use $\mathcal{R}$. In all practical code examples we inspected, executables have at least one free register available. If no such register exists, the code can be recompiled to free up one register. Whenever a write operation occurs, it is checked for safety, by seeing if the address of the write is within the range $[\mathcal{R}, OR_{max}]$, i.e., the current range for I-Log and CF-Log. If an illegal write occurs, execution is aborted and $\mathcal{V}$rf treats it as an attack. Since these "write checks" are already needed, and implemented, by *Tiny-CFA*, they can be used "as is" by *DIALED*, at no additional instrumentation cost.

4.3.4 Security Analysis

Let $\mathcal{P}$ denote an embedded operation for which control-flow and data-flow need to be attested. **P1** assures to $\mathcal{V}$rf that $\mathcal{P}$ indeed executed, and that neither its executable (including instructions added by *DIALED*'s instrumentation) nor the output (OR) produced by this execution has been tampered with. **P2** assures that all changes to the control-flow of $\mathcal{P}$ are written to OR at runtime. Similarly, **P3 & P4** guarantee that any data inputs are also logged to OR. Therefore, what we need to show is that once written, control-flow and data input values in OR can not be modified during the rest of $\mathcal{P}$ execution. This is exactly the guarantee offered by **P5**. Therefore, **P1-P5** suffice to guarantee the integrity of OR and $\mathcal{P}$'s executable (stored in ER), including I-Log and CF-Log, even in the presence of potential control-flow and data-only attacks. Given the integrity of received I-Log and CF-Log, $\mathcal{V}$rf can re-execute $\mathcal{P}$ locally and reproduce any type of runtime attack, including both control-flow and data-only attacks, that may have occurred during $\mathcal{P}$'s actual execution in $\mathcal{P}$rv.

```
1   application:
2   ; Check r4 at entry
3      cmp #OR_MAX, r4
4      jne .L11
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25      ...
```

(a) Before *DIALED* instrumentation

```
1   application:
2   ; Check r4 at entry
3      cmp #OR_MAX, r4
4      jne .L11
5   ; Save stack pointer to OR_MAX
6      mov r1, @r4
7      dec r4
8      cmp #OR_MIN, r4
9      jn .L11
10  ; Save args registers r8 - r15
11     mov r8, @r4
12     dec r4
13     cmp #OR_MIN, r4
14     jn .L11
15     mov r9, @r4
16     dec r4
17     cmp #OR_MIN, r4
18     jn .L11
19     .
20     .
21     .
22     mov r15, @r4
23     dec r4
24     cmp #OR_MIN, r4
25     jn .L11
```

(b) After *DIALED* instrumentation

Figure 4.3: Instrumentation example: Logging $\mathcal{P}$'s arguments.

4.4 *DIALED* Implementation

As described in Section 4.3.3, properties **P1, P2** and **P5** are provided by *APEX* and *Tiny-CFA*. Hence, we focus on the implementation of **P3-P4** *DIALED* instrumentation component contains about 300 lines of `Python`.

Figure 4.3 shows the instrumentation used to implement **P3** (in MSP430 Assembly) which commits $\mathcal{P}$'s arguments to I-Log. The instrumentation is added once: at the entry point of $\mathcal{P}$ to log any input parameters. Lines 2-4 are already added to $\mathcal{P}$ by *Tiny-CFA* to check whether $\mathcal{R}$ is initialized to OR_MAX. This is required by property **P5** (see Section 4.3.3). Lines 5-9 are added by *DIALED* to save the current stack pointer value to address OR_MAX. This value determines the bottom of $\mathcal{P}$'s execution stack and is used to detect and log data

```
1    ; Read from address in r15
2       mov.b @r15, r14
3
4
5
6
7
8
9
10
11
12
13
14
15                    ...
```

(a) Before *DIALED* instrumentation

```
1    ; Read from address in r15
2       mov.b @r15, r14
3    ; Compare with stack range
4       cmp r15, &OR_MAX
5       jlo .L12
6       cmp r15, r1
7       jhs .L13
8    ; Save to CF-Log if in range
9    .L12:
10      mov @r15, @r4
11      dec r4
12      cmp #OR_MIN, r4
13      jn .L11
14   .L13:
15   ...
```

(b) After *DIALED* instrumentation

Figure 4.4: Instrumentation example: Logging runtime data inputs.

inputs. Lines 10-25 record $\mathcal{P}$'s arguments (input parameters) to I-Log. In MSP430, function arguments are passed using up to 8 general-purpose registers $r8$–$r15$. Since the application defines how many arguments are passed, *DIALED* always logs such registers, to guarantee that all inputs are always captured. In this implementation, $\mathcal{R} = r4$. Hence, each register is written to the memory address pointed by $r4$. At each such write, safety checks discussed in **P5** (Section 4.3.3) are performed to assure the integrity of I-Log and CF-Log in OR. Additional checks are performed to guarantee that $\mathcal{R} = r4$ never overflows the size of OR. Such an event is treated as a security violation and reported to $\mathcal{V}$rf.

Figure 4.4 depicts the instrumentation used to log runtime data inputs to I-Log– i.e., property **P4**. Line 2 is a read instruction to copy contents from address pointed to by $r15$, to $r14$. In order to define whether this is indeed a data input, at line 4, the address in $r15$ is checked against the location of the bottom of $\mathcal{P}$'s stack, which is stored at the address of OR_MAX when $\mathcal{P}$ is invoked (lines 6-9 in Figure 4.3). Also, at line 6 in Figure 4.4, the address in $r15$ is also checked against the current stack pointer (always stored at register $r1$). If these checks fail, the value of the address pointed to by $r15$ lies outside of $\mathcal{P}$'s current execution stack: it is treated as input and committed to I-Log at line 9. Otherwise, the value is part of $\mathcal{P}$'s

current state and is not logged. Lines 10–12 check if $r4$ reached the top of OR, preventing overflows, as described in the previous paragraph.

Note that, since *DIALED* is implemented alongside *Tiny-CFA*, it cannot be abused by control flow attacks that jump in the middle of the instrumented code to skip checks and/or data input logging. Such an illegal jump is itself a control-flow change, which is committed to CF-Log by *Tiny-CFA* and thus detected by $\mathcal{V}\mathsf{rf}$.

4.5 Evaluation

We evaluate *DIALED* in terms of its hardware costs and software runtime overhead of attested embedded operations.

4.5.1 Hardware Overhead

Table 4.1 compares *DIALED* functionality and hardware costs to prior runtime attestation techniques (overviewed in Section 7.2 and 7.3). In terms of hardware, both C-FLAT [15] and OAT [124] are based on ARM TrustZone [90] which is unavailable on low-end MCUs. Atrium [135], LO-FAT [55], and LiteHAX [54] rely on dedicated (additional) hardware support from hash engines and branch-monitoring modules. Thus, their hardware overhead is far more costly than the baseline MCU (MSP430) itself. Meanwhile, *DIALED* and *Tiny-CFA* rely on low-cost hardware support of the *APEX*'s *PoX* architecture [49]. Thus, they impose much lower hardware overhead, affordable even for such low-end MCUs. Out of all other architectures, only OAT, LiteHAX, and *DIALED* provide both $\mathcal{C}\mathsf{FA}$ and $\mathcal{D}\mathsf{FA}$. Among these, *DIALED* achieves $\approx 5\times$ lower overhead in terms of combinatorial logic (Look-Up Tables – LUTs) and $\approx 50\times$ lower state hardware overhead (Registers) than the cheapest prior technique achieving both $\mathcal{C}\mathsf{FA}$ and $\mathcal{D}\mathsf{FA}$, i.e., LiteHAX.

Table 4.1: Functionality and hardware overhead comparison of existing run-time attestation architectures

Technique	**Support for $\mathcal{C}$FA**	**Support for $\mathcal{D}$FA**	**Hardware Cost – LUTs**	**Hardware Cost – Resigters**
MSP430 (baseline)	–	–	1904	691
C-FLAT	✓	–	ARM-TrustZone-M	ARM-TrustZone-M
OAT	✓	✓	ARM-TrustZone-M	ARM-TrustZone-M
Atrium	✓	–	10640 (+559%)	15960 (+2308%)
LO-FAT	✓	–	3192 (+168%)	4256 (+616%)
LiteHAX	✓	✓	1596 (+84%)	2128 (+308%)
Tiny-CFA (based on *APEX* hardware)	✓	–	302 (+16%)	44 (+6%)
DIALED (based on *APEX* hardware)	✓	✓	**302 (+16%)**	**44 (+6%)**

4.5.2 Experimental Analysis on Real-world Applications

We evaluate *DIALED* runtime overhead in three real-world applications. For the sake of fair comparison, we consider the exact same open-source applications used to evaluate *Tiny-CFA*: (1) OpenSyringePump (available at: `https://github.com/manimino/OpenSyringePump/blob/master/syringePump/syringePump.ino`) – a medical syringe pump; (2) FireSensor (available at: `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor`) – a fire detector based on temperature and humidity sensors; and (3) UltrasonicRanger (available at: `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/\Grove_Modules/ultrasonic_ranger`) – a sensor used by parking assistants for obstacle proximity measurement.

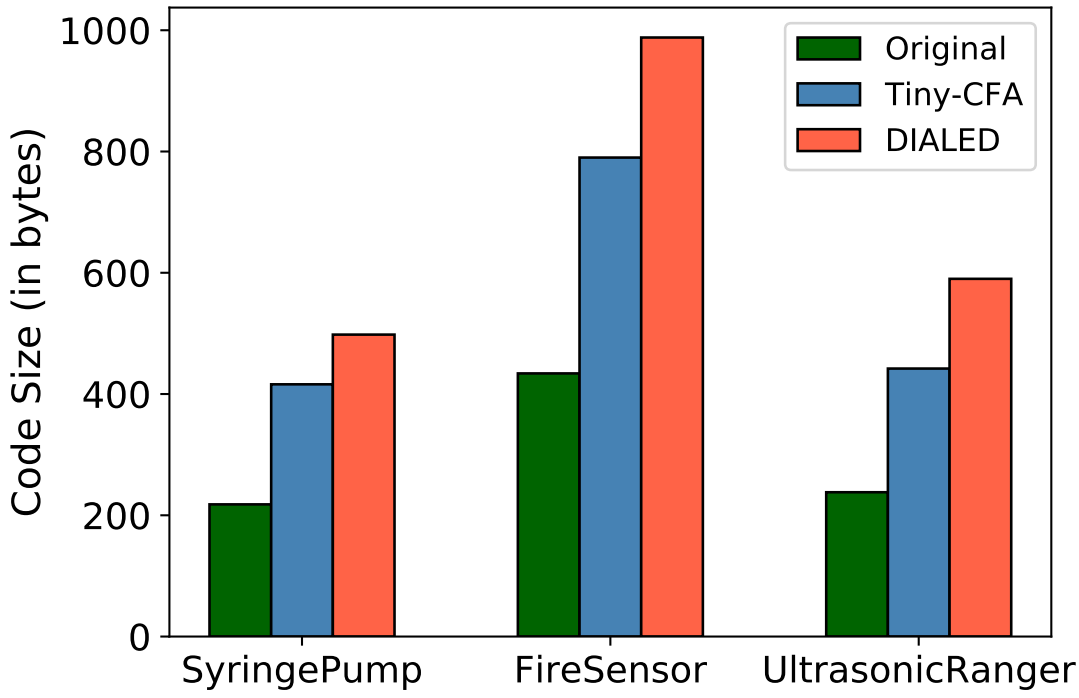

Figure 4.5: Total code size comparison

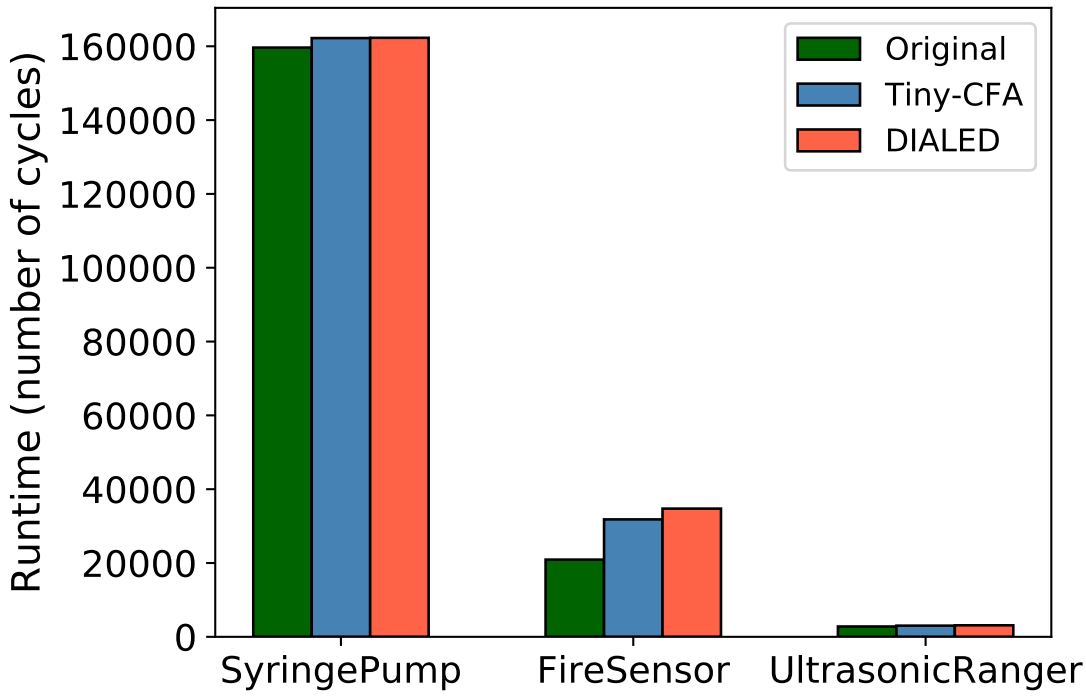

Figure 4.6: Runtime comparison

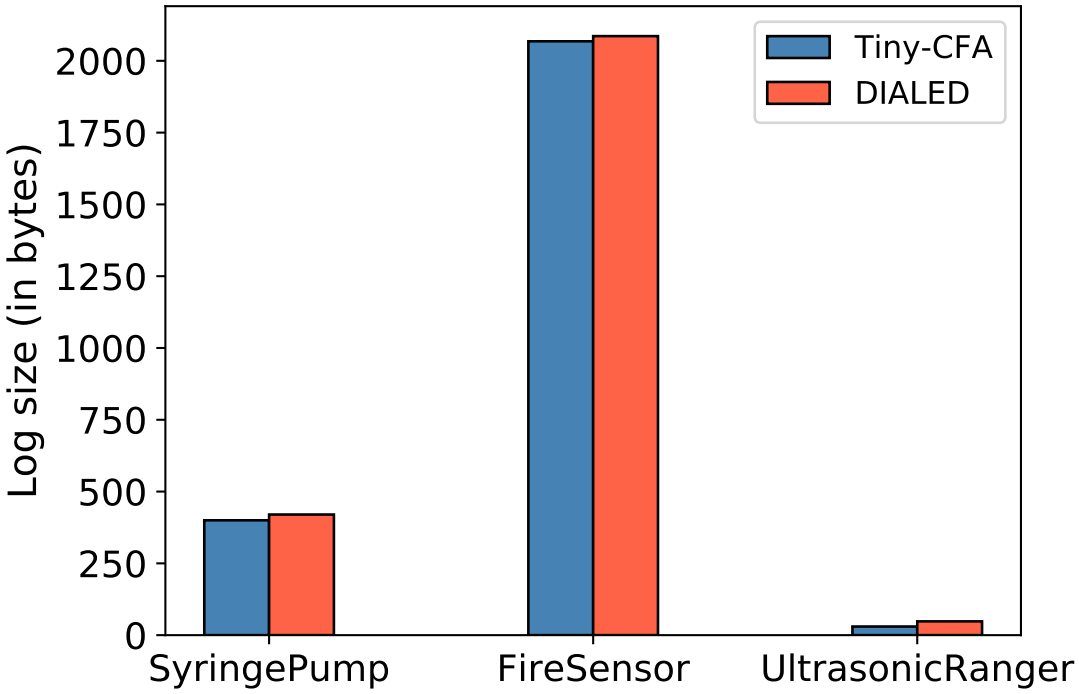

Figure 4.7: Log size comparison

We consider all sources of runtime overhead imposed by code instrumentation in these techniques: code size increase, runtime (CPU cycles), and the size of the attestation log inside OR, including I-Log and CF-Log. Figures 4.5 and 4.6 compare results of unmodified applications, with the same applications instrumented by *Tiny-CFA*, and the same applications instrumented by *DIALED*. As these results demonstrate, the overhead in both cases is dominated by the instrumentation for $\mathcal{C}$FA. On top of *Tiny-CFA*, *DIALED* code size and runtime increases range between $1\% - 20\%$. This is due to additional instructions introduced by *DIALED* instrumentation, as described in Section 4.4. Figure 4.7 shows the total OR size required to store the execution information. Recall that *DIALED* requires storage of both I-Log and CF-Log to enable detection of both control-flow and data-only attacks. Size requirements for these logs vary widely depending on the type of application (control-flow- or data-input-intensive). In general, we observe a small increase in OR size. This is due to the data input definition from Section 4.3, which allows *DIALED* to only log relevant data inputs while retaining all necessary information for $\mathcal{V}$rf's abstract execution of $\mathcal{P}$rv's

embedded operation.

Remark: we do not compare runtime overhead of *DIALED* with $\mathcal{D}$FA architectures LiteHAX and OAT, since they rely on different CPU architectures.

In summary, even though *DIALED*'s overhead is not negligible, it is well within the capabilities of low-end MCUs and suitable for practical purposes. Specifically, instrumented binary sizes are within the MCUs memory budget, its runtime is reasonable, and log sizes are small enough to fit into data memory without encroaching on the stack. We believe this to be a reasonable price for the benefit of detecting any runtime compromise in low-end MCUs.

4.6 Conclusions

We design and implement *DIALED*, the first Data-Flow Attestation ($\mathcal{D}$FA) approach targeting lowest-end MCUs. *DIALED* is composed with *Tiny-CFA*, a Control-Flow Attestation ($\mathcal{C}$FA) architecture, thus enabling detection of both control-flow and data-flow attacks at runtime. We discuss *DIALED*'s security and evaluate its performance on real embedded applications, showing that *DIALED*'s overhead is well within the capabilities of some of the most resource-constrained MCUs.

Chapter 5

Privacy-from-Birth: Protecting Sensed Data from Malicious Sensors with *VERSA*

Abstract

Many IoT devices handle sensitive and personal data. If left unprotected, ambient sensing (e.g., of temperature, audio, or video) can leak private information. At the same time, resource-constrained IoT devices have few (or no) security features.

There are many well-known techniques to secure sensed data, e.g., by authenticating communication end-points, encrypting data before transmission, and obfuscating traffic patterns. Such techniques protect sensed data from external adversaries while assuming that the sensing device itself is secure. Meanwhile, both the scale and frequency of IoT-focused attacks are growing. This prompts a natural question: *how to protect sensed data even if all software on the device is compromised?* Ideally, in order to achieve this, sensed data must be protected from its genesis, i.e., from the time when a physical analog quantity is converted into its digital counterpart and becomes accessible to software. We refer to this property as *PfB*: *Privacy-from-Birth*.

In this chapter, we formalize *PfB* and design Verified Remote Sensing Authorization (*VERSA*) – a provably secure and formally verified architecture guaranteeing that only correct execution of expected and explicitly authorized software can access and manipulate sensing interfaces, specifically, GPIO, which is the usual boundary between analog and digital worlds on IoT devices. This guarantee is obtained with minimal hardware support and holds even if all device software is compromised. *VERSA* ensures that malware can neither gain access to sensed data on the GPIO-mapped memory nor obtain any trace thereof. *VERSA* is formally verified and its open-sourced implementation targets resource-constrained IoT edge devices, commonly used for sensing. Experimental results show that *PfB* is both achievable and affordable for such devices.

Material in this chapter appeared in the Proceedings of 43rd IEEE Symposium on Security and Privacy (S&P 2022) [104].

5.1 Introduction

Over the past decade, IoT privacy issues have been recognized and explored by the research community [131, 129, 91, 138, 108]. Many techniques (e.g., [101, 85]) were developed to secure sensor data from active attacks that impersonate users, IoT back-ends, or servers. There are also techniques protecting private data from passive in-network observers that intercept traffic [127, 22, 23, 24] or perform traffic analysis based on unprotected packet headers and other metadata, e.g., sizes, timings, and frequencies. However, security of sensor data **on the device** which originates that data has not been investigated. We consider this to be a crucial issue, since all software on the device can be compromised and leak (exfiltrate) sensed data. Whereas, aforementioned techniques assume that sensing device runs the expected **benign** software.

We claim that in order to solve this problem, privacy of sensed data must be ensured "from birth". This corresponds to two requirements: (1) access to sensing interfaces must be strictly controlled, such that only authorized code is allowed to read data, (2) sensed data must be protected as soon as it is converted to digital form. Even the simplest devices (e.g., motion sensors, thermostats, and smart plugs) should be protected since prior work [39, 100, 123, 78] amply demonstrates that private – and even safety-critical – information can be inferred from sensed data. It is also well-known that even simple low-end IoT devices are subject to malware attacks. This prompts a natural question: *Can privacy of sensed data be guaranteed if the device software is compromised?* We refer to this guarantee as *Privacy-from-Birth* (*PfB*).

Some previous results considered potential software compromise in low-end devices and proposed methods to enable security services, such as remote verification of device software state (remote attestation) [62, 103, 47, 21, 30, 83], proofs of remote software execution [49], control- & data-flow attestation [54, 15, 55, 135, 124, 53, 52], as well as proofs of remote software updates, memory erasure, and system reset [48, 20, 27].

Regardless of their specifics, such techniques only detect of violations or compromises **after the fact**. In the context of *PfB*, that is too late since leakage of private sensed data likely already occurred. Notably, SANCUS [103] specifically discusses the problem of access control to sensor peripherals (e.g., GPIO) and proposes attestation of software accessing (or controlling access to) these peripherals. However, this only allows detection of compromised peripheral-accessing software and does not prevent illegal peripheral access.

To bridge this gap and obtain *PfB*, we construct the Verified Remote Sensing Authorization (*VERSA*) architecture. It provably prevents leakage of private sensor data even when the underlying device is compromised. At a high level, *VERSA* combines three key features: (1) *Mandatory Sensing Operation Authorization*, (2) *Atomic Sensing Operation Execution*, and (3) *Data Erasure on Boot* (see Section 5.3). To attain these features, *VERSA* implements a minimal and formally verified hardware monitor that runs independently from (and in parallel with) the main CPU, without modifying the CPU core. We show that *VERSA* is an efficient and inexpensive means of guaranteeing *PfB*.

5.1.1 Contributions

This chapter makes the following contributions:

- Formulates *PfB* with a high-level specification of requirements, followed by a game-based formal definition of the *PfB* goal.
- Constructs *VERSA*, an architecture that guarantees *PfB*.
- Implements and deploys *VERSA* on a commodity low-end MCU, which demonstrates its cost-effectiveness and practicality.
- Formally verifies *VERSA* implementation and proves security of the overall construction, hence obtaining provable security at both architectural and implementation levels.

VERSA implementation and its computer proofs are publicly available in [12].

5.2 Preliminaries

5.2.1 `GPIO` & MCU Sensing

A GPIO port is a set of GPIO pins arranged and controlled together, as a group. The MCU-addressable memory for a GPIO port is physically mapped (hard-wired) to physical ports that can be connected to a variety of external circuits, such as analog sensors and actuators, as shown in Figure 2.1. Each GPIO pin can be set to function as either an input or output, hence called "general purpose". Input signals produced by external circuits can be obtained by the MCU software by reading from GPIO-mapped memory. Similarly, egress electric signals (high or low voltage) can be generated by the MCU software by writing (logical 1 or 0) to GPIO-mapped memory.

Remark: "GPIO-mapped memory" includes the set of all software-readable memory regions connected to external sensors. In some cases, this set may even include multiple physical memory regions for a single physical pin. For instance, if a given GPIO pin is also equipped with an Analog-to-Digital Converter (ADC), a GPIO input could be reflected on different memory regions depending on whether the ADC is active or inactive. All such regions are considered "GPIO-mapped memory" and we refer to it simply as `GPIO`. Using this definition, in order to access sensor data, software running on the MCU must read from `GPIO`.

We also note that various applications require different sensor regimes [6]: event-driven, periodic, and on-demand. Event-driven sensors report sensed data when a trigger event occurs, while periodic sensors report sensor data at fixed time intervals. On-demand (or query-driven) sensors report sensor data whenever requested by an external entity. Although

we initially consider on-demand sensing, as discussed in Section 5.3, the proposed design is applicable to other regimes.

5.2.2 LTL, Model Checking, & Verification

Our verification and proof methodologies are in-line with prior work on the design and verification of security architectures proving code integrity and execution properties for the same class of MCUs [47, 49, 51, 50, 18]. However, to the best of our knowledge, no prior work tackled formal models and definitions, or designed services, for guaranteed sensed data privacy. This section overviews our verification and proof methodologies that allow us to later show that *VERSA* achieves required *PfB* properties and end-goals.

Computer-aided formal verification typically involves three steps. First, the system of interest (e.g., hardware, software, or 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 done via Theorem Proving [93] or Model Checking [41]. 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: (σ, σ_0, T), where σ is the finite set of states, $\sigma_0 \subseteq \sigma$ is the set of possible initial states, and $T \subseteq \sigma \times \sigma$ is the transition relation set, which describes the set of states that can be reached in a single step from each state. Such usage of LTL allows for representing a system behavior over time.

- $\mathbf{X}\phi$ – ne<u>X</u>t ϕ: holds if ϕ is true at the next system state.
- $\mathbf{G}\phi$ – <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: ϕ $\mathbf{W}$ $\psi \equiv (\phi$ $\mathbf{U}$ $\psi) \vee$ $\mathbf{G}(\phi)$.
- ϕ $\mathbf{B}$ ψ – ϕ <u>B</u>efore ψ: holds if the existence of state where ψ holds implies the existence of at least one earlier state where ϕ holds. Equivalently: ϕ $\mathbf{B}$ $\psi \equiv \neg(\neg\phi$ $\mathbf{U}$ $\psi)$.

Figure 5.1: LTL Quantifiers

Our verification strategy benefits from the popular model checker NuSMV [40], which can verify generic hardware or software 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 [76] as the standard RTL design already relies on describing hardware as FSMs. LTL specifications are 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 chapter, we are interested in the LTL quantifiers shown in Figure 5.1.

NuSMV works by exhaustively enumerating all possible states of a given system FSM and by checking each state against LTL specifications. If any desired specification is found not to hold for specific states (or transitions between states), the model checker provides a trace that leads to the erroneous state, which helps correct the implementation accordingly. As a consequence of exhaustive enumeration, proofs for complex systems that involve complex properties often do not scale due to the so-called "state explosion" problem. To cope with it, our verification approach is to specify smaller LTL sub-properties separately and verify each respective hardware sub-module for compliance. In this process, our verification pipeline automatically converts digital hardware, described at RTL using Verilog, to Symbolic Model Verifier (SMV) [97] FSMs using Verilog2SMV [76]. The SMV representation is then fed

to NuSMV for verification. Then, the composition of LTL sub-properties (verified in the model-checking phase) is proven to achieve a desired end-to-end implementation goal, also specified in LTL. This step uses an LTL theorem prover [60].

In our case, we show that the end-to-end goal of *VERSA*, in composition with *VRASED*, is sufficient to achieve *PfB* via cryptographic reduction from the formal security definition of *VRASED*. These steps are discussed in detail in Section 5.7.

5.3 *VERSA* Overview

VERSA involves two entities: a trusted remote controller ($\mathcal{C}$trl) and a device ($\mathcal{D}$ev). We expect $\mathcal{C}$trl to be a relatively powerful computing entity, e.g., a home gateway, a backend server or even a smartphone. *VERSA* protects sensed data on $\mathcal{D}$ev by keeping it (and any function thereof) confidential. This implies: (1) controlling GPIO access by blocking attempted reads by unauthorized software, and (2) keeping execution traces (i.e., data allocated by GPIO-authorized software) confidential. Therefore, access to GPIO is barred by default. GPIO is unlocked only for benign binaries that are pre-authorized by $\mathcal{C}$trl. Whenever a binary is deemed to be authorized on $\mathcal{D}$ev, *VERSA* creates for it an ephemeral isolated execution environment and permits its one-time execution. This isolated environment lasts until execution ends, which corresponds to reaching the legal exit point of the authorized binary. Therefore, by including a clean-up routine immediately before the legal exit, we can assure that all execution traces, including all sensitive information, are erased. Any attempt to interrupt, or tamper with, isolated execution causes an immediate system-wide reset, which erases all data traces.

We use the term "Sensing Operation", denoted by $\mathcal{S}$, to refer to a self-contained and logically independent binary (e.g., a function) that is responsible for processing data obtained through

one or more reads from GPIO.

VERSA achieves *PfB* via three key features:

[A] *Mandatory Sensing Operation Authorization* requires explicit authorization issued by $\mathcal{C}$trl before any $\mathcal{D}$ev software reads from GPIO. Recall that access to GPIO is blocked by default. Each authorization token (ATok) coming from $\mathcal{C}$trl allows one execution of a specific sensing operation $\mathcal{S}$, although a single execution of $\mathcal{S}$ can implement several GPIO reads. ATok has the following properties:

1. It can be authenticated by $\mathcal{D}$ev as having been issued by $\mathcal{C}$trl; this includes freshness;
2. It grants privileges **only to a specific** $\mathcal{S}$ to access GPIO during its execution; and
3. It can only be used once.

$\mathcal{C}$trl can authorize multiple executions of $\mathcal{S}$ by issuing a batch of tokens, i.e., $\mathsf{ATok}_1, ..., \mathsf{ATok}_n$, for up to n executions of $\mathcal{S}$. Although supporting multiple tokens is unnecessary for on-demand sensing, it might be useful for periodic or event-driven sensing regimes discussed in Section 5.2.1.

[B] *Atomic Sensing Operation Execution* ensures that, once authorized by $\mathcal{C}$trl, $\mathcal{S}$ is executed with the following requirements:

1. $\mathcal{S}$ execution starts from its legal entry point (first instruction) and runs until its legal exit point (last instruction). This assumes a single pair of entry-exit points;
2. $\mathcal{S}$ execution can not be interrupted and its intermediate results cannot be accessed by external means, e.g., via DMA controllers; and
3. An immediate MCU reset is triggered if either (1) or (2) above is violated.

[C] *Data Erasure on Reset/Boot* works with **[B]** to guarantee that, sensed data (or any function thereof) obtained during $\mathcal{S}$ execution is not leaked due to errors or violations of security properties, which cause MCU reset per item (3) above. This feature must guarantee that all values that remain in RAM after a hard reset and the subsequent boot process, are erased before any unprivileged software can run. While some architectures already provide memory erasure on boot, for those MCUs that do not do so, it can be obtained by calling a secure RAM erasure function at boot time, e.g., as a part of a ROM-resident bootloader code. Section 5.10.2 discusses this further.

At a high level, correct implementation of the aforementioned three features suffices to obtain *PfB*, because:

- Any compromised/modified binary can not access `GPIO` since it has no authorization from $\mathcal{C}$trl.
- Any authorized binary $\mathcal{S}$ must be invoked properly and run atomically, from its first, and until its last, instruction.
- Since $\mathcal{S}$ is invoked properly, the intended behavior of $\mathcal{S}$ is preserved. Code reuse attacks are not possible unless they occur as a result of bugs in $\mathcal{S}$ implementation itself. $\mathcal{C}$trl can always check for such bugs in $\mathcal{S}$ prior to authorization; see Section 5.5.2.
- $\mathcal{S}$ runs uninterrupted, meaning that it can erase all traces of its own execution from the stack before passing control to unprivileged applications. This guarantees that no sensor data remains in memory when $\mathcal{S}$ terminates.
- *VERSA* assures that any violation of the aforementioned requirements causes an MCU reset, triggering erasure of all data memory. Therefore, malware that attempts to interrupt $\mathcal{S}$ before completion, or tamper with $\mathcal{S}$ execution integrity, will cause all data used by $\mathcal{S}$ to be erased.

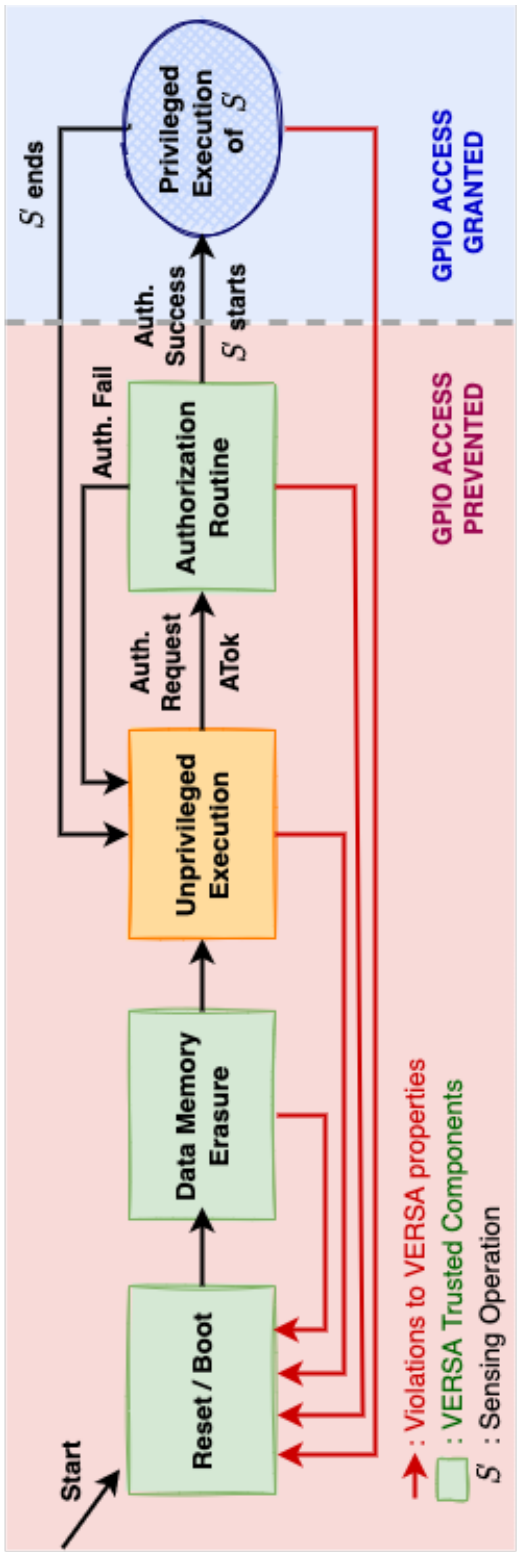

Figure 5.2: MCU execution workflow with *VERSA*.

Support for Output Encryption: $\mathcal{S}$ might process and use sensor data locally as part of its own execution, or generate some output that needs to be returned to $\mathcal{C}$trl. In the latter case, encryption of $\mathcal{S}$ output is necessary. For this reason, *VERSA* supports the generation of a fresh key derived from `ATok` (thus implicitly shared between $\mathcal{C}$trl and $\mathcal{S}$). This key is only accessible to $\mathcal{S}$ during authorized execution. Hence, $\mathcal{S}$ can encrypt any data to be exported with this key and ensure that encrypted results can only be decrypted by $\mathcal{C}$trl.

Since we assume that the encryption function is part of $\mathcal{S}$, it cannot be interrupted (or tampered with) by any unprivileged software or external means. Importantly, the encryption key is only accessible to $\mathcal{S}$ (similar to `GPIO`) and shielded from all other software. Furthermore, the choice of the encryption algorithm is left up to the specific $\mathcal{S}$ implementation.

Figure 5.2 illustrates MCU execution workflow discussed in this section.

5.4 MCU Machine Model

5.4.1 Execution Model

To enable formal specification of *PfB* guarantees, we formulate the MCU execution model in Definition 5.1. It represents MCU operation as a discrete sequence of MCU states, each corresponding to one clock cycle – the smallest unit of time in the system. We say that the subsequent MCU state is defined based on the current MCU state (which includes current values in memory/registers, as well as any hardware signals and effects, such as external inputs, actions by DMA controller(s), and interrupts) and the current instruction being executed by the CPU core. Similarly, the instruction to be executed in the next state is determined by the current state and the current instruction being executed.

For example, an arithmetic instruction (e.g., `add` or `mult`) causes the program counter (PC)

Definition 5.1 (MCU Execution Model)**.**
1 – Execution *is modeled as a sequence of MCU states* $\mathsf{S} := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$ *and a sequence of instructions* $\mathsf{I} := \{\mathsf{i}_0, ..., \mathsf{i}_n\}$*. Since the next MCU state and the next instruction to be executed are determined by the current MCU state and the current instruction being executed, these discrete transitions are denoted as shown in the following example:*

$$(\mathsf{s_1}, \mathsf{i}_j) \leftarrow \boldsymbol{EXEC}(\mathsf{s_0}, \mathsf{i_0}); \quad (\mathsf{s_2}, \mathsf{i}_k) \leftarrow \boldsymbol{EXEC}(\mathsf{s_1}, \mathsf{i}_j); \quad \ldots \quad (\mathsf{s}_m, \perp) \leftarrow \boldsymbol{EXEC}(\mathsf{s}_{m-1}, \mathsf{i}_l)$$

The sequence I *represents the physical order of instructions in memory, which is not necessarily the order of their execution. The next instruction and state are also affected by current external inputs, current data-memory values, and current hardware events, e.g., interrupts or resets, which are modeled as properties of each execution state in* S*. The MCU always starts execution (at boot or after a reset) from state* s_0 *and initial instruction* i_0*. EXEC produces* $\perp$ *as the next instruction if there is no instructions left to execute.*

2 – State Properties as Sets: *sets are used to model relevant execution properties and characterize effects/actions occurring within a given state* s_t*. We are particularly interested in the behaviors corresponding to the following sets:*

1. ***READ:*** *all states produced by the execution of an instruction* i *that reads the value from memory to a register.*
2. ***WRITE:*** *all states produced by the execution of an instruction* i *that writes the value from a register to memory.*
3. $\boldsymbol{DMA^R}$***:*** *all states produced as a result of DMA reading from memory.*
4. $\boldsymbol{DMA^W}$***:*** *all states* s_t *produced as a result of DMA writing to memory.*
5. ***IRQ:*** *all states* s_t *where an interrupt is triggered.*
6. ***RESET:*** *all states* s_t *wherein an MCU reset is triggered.*

Note that these sets are not disjoint, i.e., s_t *can belong to multiple sets. Also, the aforementioned sets do not aim to model all possible MCU behaviors, but only the ones relevant to PfB. Finally, we further subdivide sets that model memory access into subsets relating to memory regions of interest. For example, considering a contiguous memory region* $\mathcal{M} = [\mathcal{M}_{min}, \mathcal{M}_{max}]$*,* $READ_{\mathcal{M}}$ *is a subset of READ containing only the states produced through EXEC of instructions that read from the memory region* $\mathcal{M}$*. We use the same notation to refer to other subsets, e.g.,* $WRITE_{\mathcal{M}}$*,* $DMA^R_{\mathcal{M}}$*, and* $DMA^W_{\mathcal{M}}$*.*

Figure 5.3: MCU Execution Model

Definition 5.2 (Hardware Model).
$\mathcal{M}$ denotes a contiguous memory region within addresses $\mathcal{M}_{min}$ and $\mathcal{M}_{max}$ in physical memory of $\mathcal{D}$ev, i.e., $\mathcal{M} := [\mathcal{M}_{min}, \mathcal{M}_{max}]$.
s represents the system execution state at a given CPU cycle.

Program counter & instruction execution:

$$\mathbf{G}: \{[\mathbf{X}(\mathsf{s}) \leftarrow EXEC(\mathsf{s}, \mathsf{i}_k) \wedge \mathsf{i}_k \in \mathcal{M})] \rightarrow (PC \in \mathcal{M})\} \tag{5.1}$$

Memory Reads/Writes:

$$\mathbf{G}: \{\mathbf{X}(\mathsf{s}) \in READ_{\mathcal{M}} \rightarrow (R_{en} \wedge D_{addr} \in \mathcal{M})\} \tag{5.2}$$

$$\mathbf{G}: \{\mathbf{X}(\mathsf{s}) \in WRITE_{\mathcal{M}} \rightarrow (W_{en} \wedge D_{addr} \in \mathcal{M})\} \tag{5.3}$$

$$\mathbf{G}: \{(\mathbf{X}(\mathsf{s}) \in DMA^{R}_{\mathcal{M}} \vee \mathbf{X}(\mathsf{s}) \in DMA^{W}_{\mathcal{M}}) \rightarrow (DMA_{en} \wedge DMA_{addr} \in \mathcal{M})\} \tag{5.4}$$

Interrupts (irq) and Resets:

$$\mathbf{G}: \{\mathsf{s} \in IRQ \leftrightarrow irq\} \tag{5.5}$$

$$\mathbf{G}: \{\mathsf{s} \in RESET \leftrightarrow reset\} \tag{5.6}$$

Figure 5.4: MCU Hardware Model

to point to the subsequent address in physical memory. However, an interrupt (which is a consequence of the current MCU state) may occur and deviate from the normal execution flow. Alternatively, a branching instruction may be executed and cause PC to jump to some arbitrary instruction that is not necessarily located at the subsequent position in the MCU flash memory.

In order to reason about events during the MCU operation, we say that each MCU state can belong to one or more sets. Belonging to a given set implies that the state has a given property of interest. Definition 5.1 introduces six sets of interest, representing states in which memory is read/written by CPU or DMA, as well as states in which an interrupt or reset occurs.

5.4.2 Hardware Signals

We now formalize the effects of execution, modeled in Definition 5.1, to the values of concrete hardware signals that can be monitored by *VERSA* hardware in order to attain *PfB*

guarantees. Informally, we model the following simple axioms:

[A1] PC: contains the memory address containing the instruction being executed at a given cycle.

[A2] CPU Memory Access: Whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. A data read-enable bit (R_{en}) must be set for a read access and a data write-enable bit (W_{en}) must be set for a write access.

[A3] DMA: Whenever a DMA controller attempts to access the main memory, a DMA-address signal (DMA_{addr}) contains the address of the accessed memory location and a DMA-enable bit (DMA_{en}) must be set.

[A4] Interrupts: When hardware interrupts or software interrupts happen, the irq signal is set.

[A5] MCU reset: At the end of a successful reset routine, all registers (including PC) are set to zero before restarting software execution. The reset handling routine cannot be modified, as resets are handled by MCU in hardware. When a reset happens, the corresponding **Reset** signal is set. The same signal is also set when the MCU initializes for the first time.

This model strictly adheres to MCU specifications, assumed to be correctly implemented by the underlying MCU core.

Definition 5.2 presents formal specifications for the aforementioned axioms in LTL. Instead of explicitly quantifying time, LTL embeds time within the logic by using temporal quantifiers (see Section 5.2). Hence, rather than referring to execution states using temporal variables (i.e., state t, state $t + 1$, state $t + 2$), a single variable (s) and LTL quantifiers suffice to

Definition 5.3 (Syntax: *PfB* scheme)**.**

A Privacy-from-Birth (PfB) scheme is a tuple of algorithms [Authorize, Verify, XSensing]*:*

1. $\mathsf{Authorize}^{\mathcal{C}\mathsf{trl}}(\mathcal{S}, \cdots)$*: an algorithm executed by* $\mathcal{C}\mathsf{trl}$ *taking as input at least one executable* $\mathcal{S}$ *and producing at least one authorization token* `ATok` *which can be sent to* $\mathcal{D}\mathsf{ev}$ *to authorize one execution of* $\mathcal{S}$ *with access to* `GPIO`*.*

2. $\mathsf{Verify}^{\mathcal{D}\mathsf{ev}}(\mathcal{S}, \texttt{ATok}, \cdots)$*: an algorithm (with possible hardware-support), executed by* $\mathcal{D}\mathsf{ev}$*, that takes as input* $\mathcal{S}$ *and* `ATok`*. It uses* `ATok` *to check whether* $\mathcal{S}$ *is pre-authorized by* $\mathcal{C}\mathsf{trl}$ *and outputs* $\top$ *if verification succeeds, and* $\bot$ *otherwise.*

3. $\mathsf{XSensing}^{\mathcal{D}\mathsf{ev}}(\mathcal{S}, \cdots)$*: an algorithm (with possible hardware-support) that executes* $\mathcal{S}$ *in* $\mathcal{D}\mathsf{ev}$*, producing a sequence of states* $E := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$*. It returns* $\top$*, if sensing successfully occurs during* $\mathcal{S}$ *execution, i.e.,* $\exists \mathsf{s} \in E$ *such that* $(\mathsf{s} \in READ_{\texttt{GPIO}}) \wedge (\mathsf{s} \notin RESET)$*); it returns* $\bot$*, otherwise.*

Remark: *In the parameter list,* $(\cdots)$ *means that additional/optional parameters might be included depending on the specific PfB construction.*

Figure 5.5: Syntax of the *PfB* Scheme

specify, e.g., "current", "next", "future" system states (s). For this part of the model, we are mostly interested in: (1) describing MCU state at the next CPU cycle ($\mathbf{X}(\mathsf{s})$) as a function of the MCU state at the current CPU cycle (s), and (2) describing which particular MCU signals must be triggered in order for $\mathbf{X}(\mathsf{s})$ to be in each of the sets defined in Definition 5.1.

LTL statements in Definition 5.2 formally model axioms **[A1]-[A5]**, i.e., the subset of MCU behavior that is relevant to, and sufficient for formally verifying, *VERSA*. LTL (5.1) models **[A1]**, (5.2) and (5.3) model **[A2]**, and each (5.4), (5.5), and (5.6) models **[A3]**, **[A4]**, and **[A5]**, respectively.

5.5 *PfB* Definitions

Based on the specified machine model, we now proceed with the formal definition of *PfB*.

DEFINITION 5.4 (*PfB* Game-based Definition)**.**

5.4.1 Auxiliary Notation & Predicate(s):

- *Let* $\mathcal{K}$ *be a secret string of bit-size* $|\mathcal{K}|$*; and* λ *be the security parameter, determined by* $|\mathcal{K}|$*, i.e.,* $\lambda = \Theta(|\mathcal{K}|)$*;*

- *Let* $\mathsf{atomicExec}$ *be a predicate evaluated on some sequence of states* $\boldsymbol{S}$ *and some software – i.e., some sequence of instructions* $\boldsymbol{I}$*.*
  $\mathsf{atomicExec}(\boldsymbol{S} := \{\mathsf{s}_1, ..., \mathsf{s}_m\}, \boldsymbol{I} := \{\mathsf{i}_0, ..., \mathsf{i}_n\}) \equiv \top$ *if and only if the following hold; otherwise,* $\mathsf{atomicExec}(\boldsymbol{S}, \boldsymbol{I}) \equiv \bot$*.*

  1. ***Legal Entry Instruction:*** *The first execution state* s_1 *in* $\boldsymbol{S}$ *is produced by the execution of the first instruction* i_0 *in* $\boldsymbol{I}$*.*
     i.e., $(\mathsf{s}_1 \leftarrow EXEC(\mathsf{i}_0, \mathsf{s}*)) \ \vee \ (\mathsf{s}_1 \in RESET)$*, where* $\mathsf{s}*$ *is any state prior to* s_1*.*
  2. ***Legal Exit Instruction:*** *The last execution state* s_m *in* $\boldsymbol{S}$ *is produced by the execution of the last instruction* i_n *in* $\boldsymbol{I}$*.*
     i.e., $(\mathsf{s}_m \leftarrow EXEC(\mathsf{i}_n, \mathsf{s}_{m-1})) \ \vee \ (\mathsf{s}_m \in RESET)$*.*
  3. ***Self-Contained Execution:*** *For all* s_j *in* $\boldsymbol{S}$*,* s_j *is produced by the execution of an instruction* i_k *in* $\boldsymbol{I}$*, for some* k*.*
     i.e., $(\mathsf{s}_j \leftarrow EXEC(\mathsf{i}_k, \mathsf{s}_{j-1})) \ \vee \ (\mathsf{s}_j \in RESET)$*, for some* $\mathsf{i}_k \in \boldsymbol{I}$*.*
  4. ***No Interrupts, No DMA:*** *For all* s_j *in* $\boldsymbol{S}$*,* s_j *is neither in the IRQ or DMA. i.e.,* $[(\mathsf{s}_j \notin IRQ) \wedge (\mathsf{s}_j \notin DMA)] \ \vee \ (\mathsf{s}_j \in RESET)$*.*

5.4.2 *PfB-**Game:*** *The challenger plays the following game with* $\mathcal{A}\mathsf{dv}$*:*

1. $\mathcal{A}\mathsf{dv}$ *is given full control over* $\mathcal{D}\mathsf{ev}$ *software state, implying* $\mathcal{A}\mathsf{dv}$ *can execute any (polynomially sized) sequence of arbitrary instructions* $\{\mathsf{i}_0^{\mathcal{A}\mathsf{dv}}, ..., \mathsf{i}_n^{\mathcal{A}\mathsf{dv}}\}$*, inducing the associated changes in* $\mathcal{D}\mathsf{ev}$*'s sequence of execution states;*

2. $\mathcal{A}\mathsf{dv}$ *has oracle access to polynomially many calls to* Verify*.* $\mathcal{A}\mathsf{dv}$ *also has access to the set of software executables,* $\mathsf{SW} := \{S_1, ..., S_l\}$*, and the set of all corresponding authorization "tokens",* $\mathsf{T} := \{\boldsymbol{ATok}_1, ..., \boldsymbol{ATok}_l\}$*, ever produced by any prior* $\mathcal{C}\mathsf{trl}$ *calls to* $\mathsf{Authorize}$ *up until time* t*. i.e.,* $\boldsymbol{ATok}_j \leftarrow \mathsf{Authorize}(\mathcal{S}_j, ...)$*, for all* j*.*

3. *Let* $\mathsf{U} \subset \mathsf{T}$ *be the set of all "used" authorization tokens up until time* t*, i.e.,* $\boldsymbol{ATok}_j \in \mathsf{U}$*, if a call to* $\mathsf{XSensing}(\mathcal{S}_j, ...)$ *returned* $\top$ *up until time* t*; Let* P *be the set of "pending" (issued but not used) authorization tokens, i.e.,* $\mathsf{P} := \mathsf{T} \setminus \mathsf{U}$*.*

4. *At any arbitrary time* t*,* $\mathcal{A}\mathsf{dv}$ *wins if it can perform an* ***unauthorized or tampered sensing execution****, i.e.:*

   - $\mathcal{A}\mathsf{dv}$ *triggers an* $\mathsf{XSensing}(\mathcal{S}_{\mathcal{A}\mathsf{dv}}, ...)$ *operation that returns* $\top$*, for* $\forall \mathcal{S}_{\mathcal{A}\mathsf{dv}} \notin \mathsf{SW}$*, or*
   - $\mathcal{A}\mathsf{dv}$ *triggers* $(\boldsymbol{S}, \top) \leftarrow \mathsf{XSensing}(\mathcal{S}_j, ...)$ *such that* $\mathsf{atomicExec}(\boldsymbol{S}, \mathcal{S}_j) \equiv \bot$*, for some* $\mathcal{S}_j \in \mathsf{SW}$ *and* $\boldsymbol{ATok}_j \in \mathsf{U}$ *.*

5.4.3 *PfB-**Security:*** *A scheme is considered PfB-Secure iff, for all PPT adversaries* $\mathcal{A}\mathsf{dv}$*, there exists a negligible function* $\mathsf{negl}[]$ *such that:*

$$Pr[\mathcal{A}\mathsf{dv}, \textit{PfB-Game}] \leq \mathsf{negl}(\mathsf{l})$$

Figure 5.6: *PfB* Security Game

5.5.1 *PfB* Syntax

A *PfB* scheme involves two parties: $\mathcal{C}$trl and $\mathcal{D}$ev. $\mathcal{C}$trl authorizes $\mathcal{D}$ev to execute some software $\mathcal{S}$ which accesses GPIO. It should be impossible for any software different from $\mathcal{S}$ to access GPIO data, or any function thereof (see Definition 5.4). $\mathcal{C}$trl is trusted to only authorize functionally correct code. The goal of a *PfB* scheme is to facilitate sensing-dependent execution while keeping all sensed data private from all other software.

Definition 5.3 specifies a syntax for *PfB* scheme composed of three functionalities: Authorize, Verify, and XSensing. Authorize is invoked by $\mathcal{C}$trl to produce an authorization token, ATok, to be sent to $\mathcal{D}$ev, enabling $\mathcal{S}$ to access GPIO. Verify is executed at $\mathcal{D}$ev with ATok as input, and it checks whether ATok is a valid authorization for the software on $\mathcal{D}$ev. If and only if this check succeeds, Verify returns $\top$. Otherwise, it returns $\bot$. The verification success indicates one execution of $\mathcal{S}$ granted on $\mathcal{D}$ev via XSensing. XSensing is considered successful (returns $\top$), if there is at least one MCU state produced by XSensing where a GPIO read occurs without causing an MCU *reset*, i.e., $(\mathsf{s} \in \mathit{READ}_{\mathit{GPIO}}) \wedge \neg(\mathsf{s} \in \mathit{RESET})$. Otherwise, XSensing returns $\bot$. That is, XSensing models execution of any software in the MCU and its return symbol indicates whether a GPIO read occurred during its execution. Therefore, invocation of XSensing on any input software that does not read from GPIO returns $\bot$. Figure 5.7 illustrates a benign *PfB* interaction between $\mathcal{C}$trl and $\mathcal{D}$ev.

5.5.2 Assumptions & Adversarial Model

We consider an adversary, $\mathcal{A}$dv, that controls the entire software state of $\mathcal{D}$ev, including PMEM (flash) and DMEM (DRAM). It can attempt to modify any writable memory (including PMEM) or read any memory, including peripheral regions, such as GPIO, unless explicitly protected by verified hardware. It can launch code injection attacks to execute arbitrary instructions from PMEM or even DMEM (if the MCU architecture supports execu-

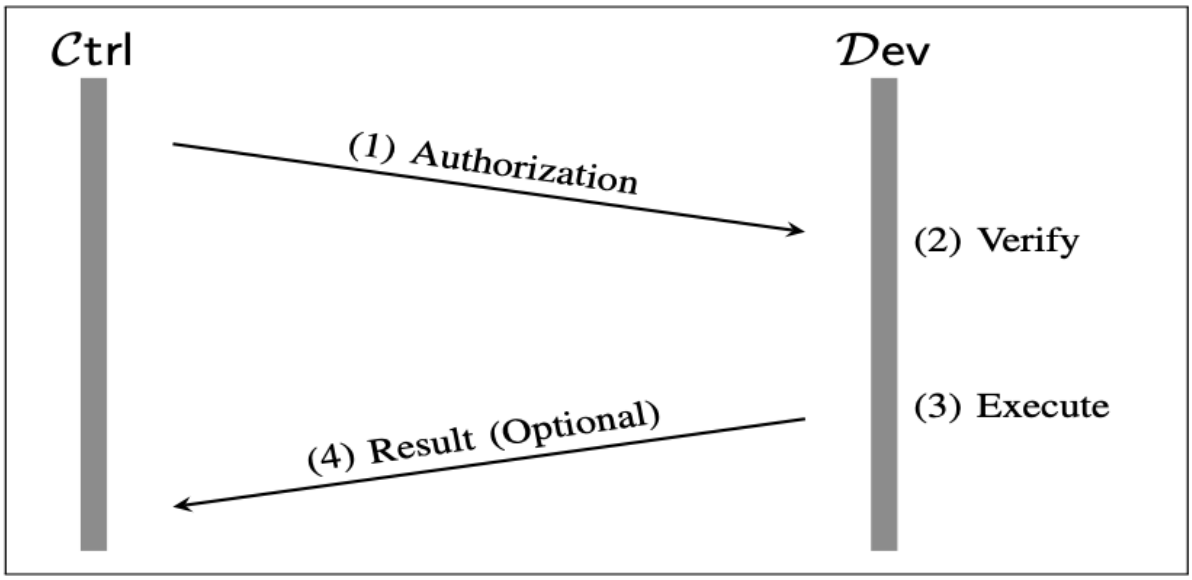

Figure 5.7: *PfB* interaction between $\mathcal{C}$trl and $\mathcal{D}$ev

tion from DMEM). It also has full control over any DMA controllers on $\mathcal{D}$ev that can directly read/write to any part of the memory independently of the CPU. It can induce interrupts to pause any software execution and leak information from its stack, or change its control-flow. We consider Denial-of-Service (DoS) attacks, whereby $\mathcal{A}$dv abuses *PfB* functionality in order to render $\mathcal{D}$ev unavailable, to be out-of-scope. These are attacks on $\mathcal{D}$ev availability and not on sensed data privacy.

Executable Correctness: we stress that *VERSA* aims to guarantee that $\mathcal{S}$, as specified by $\mathcal{C}$trl, is the only software that can access and process `GPIO` data. Similar to other trusted hardware architectures, *PfB* does not check for lack of implementation bugs within $\mathcal{S}$; thus it is not concerned with run-time (e.g., control-flow and data-only) attacks. As a relatively powerful and trusted entity $\mathcal{C}$trl can use various well-known vulnerability detection methods, e.g., fuzzing [38], static analysis [44], and even formal verification, to scrutinize $\mathcal{S}$ before authorizing it.

Physical Attacks: Physical and hardware-focused attacks are considered out of scope.

An $\mathcal{A}$dv that is physically present near $\mathcal{D}$ev can indeed sense what $\mathcal{D}$ev reads through its sensor peripherals; so privacy concerns may not be applicable in this case. We also assume that $\mathcal{A}$dv cannot modify code in ROM, induce hardware faults, or retrieve $\mathcal{D}$ev's secrets via side-channels that require $\mathcal{A}$dv's physical presence. Protection against such attacks can be obtained via standard physical security techniques [110]. This assumption is in line with related work on trusted hardware architectures for embedded systems [62, 47, 83, 30].

5.5.3 *PfB* Game-based Definition

Definition 5.4 starts by introducing an auxiliary predicate atomicExec. It defines whether a particular sequence of execution states (produced by the execution of some software $\mathcal{S}$) adheres to all necessary execution properties for *Atomic Sensing Operation Execution* discussed in Section 5.3.

In atomicExec (in Definition 5.4.1), conditions 1-3 guarantee that a given $\mathcal{S}$ is executed as a whole and no external instruction is executed between its first and last instructions. Condition 4 assures that DMA is inactive during execution, hence protecting intermediate variables in DMEM against DMA tampering. Additionally, malicious interrupts could be leveraged to illegally change the control-flow of $\mathcal{S}$ during its execution. Therefore, condition 4 stipulates that both cases cause atomicExec to return $\perp$.

PfB-Game in Definition 5.4.2 models $\mathcal{A}$dv's capabilities by allowing it to execute any sequence of (polynomially many) instructions. This models $\mathcal{A}$dv's full control over software executed on the MCU, as well as its ability to use software to modify memory at will. It can also call Verify any (polynomial) number of times in an attempt to gain an advantage (e.g., learn something) from Verify executions.

To win the game, $\mathcal{A}$dv must succeed in executing some software that does not cause an MCU

reset, and either: (1) is unauthorized, yet reads from GPIO, or (2) is authorized, yet violates atomicExec predicate conditions during its execution.

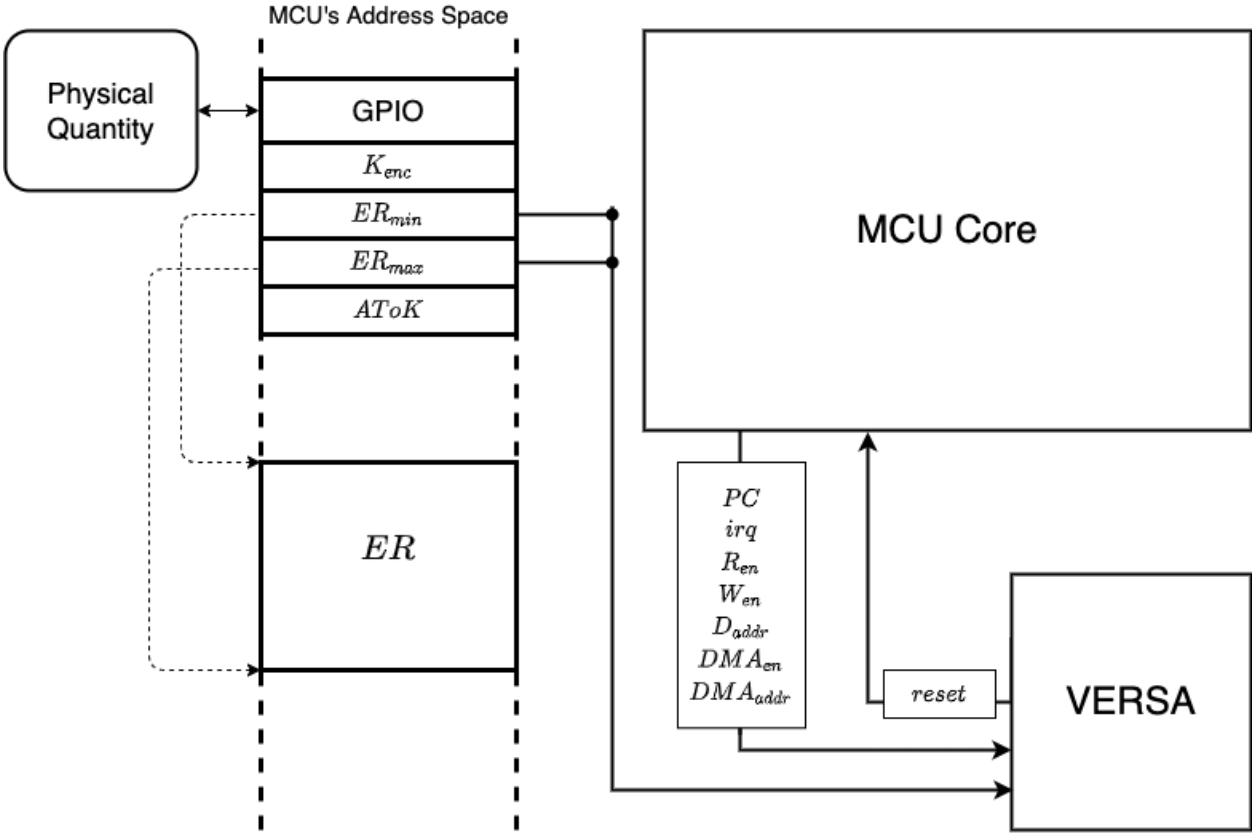

Figure 5.8: *VERSA* Architecture

5.6 *VERSA*: Realizing *PfB*

VERSA runs in parallel with the MCU core and monitors a set of MCU signals: PC, D_{addr}, R_{en}, W_{en}, DMA_{en}, DMA_{addr}, and irq. It also monitors ER_{min} and ER_{max}, the boundary memory addresses of ER where $\mathcal{S}$ is stored; these are collectively referred to as "*METADATA*". *VERSA* hardware module detects privacy violations in real-time, based on aforementioned signals and *METADATA* values, causing an immediate MCU reset. Figure 5.8 shows the *VERSA* architecture. For quick reference, MCU signals and memory regions relevant to *VERSA* are summarized in Table 5.1. To facilitate the specification of *VERSA*

Table 5.1: Notation Summary

Notation	Description
PC	Current program counter value
R_{en}	1-bit signal that indicates if MCU is reading from memory
W_{en}	1-bit signal that indicates if MCU is writing to memory
D_{addr}	Memory address of an MCU memory access
DMA_{en}	1-bit signal that indicates if DMA is active
DMA_{addr}	Memory address being accessed by DMA, when active
irq	1-bit signal that indicates if an interrupt is happening
Reset	Signal that reboots the MCU when set to logic '1'
ER	A configurable memory region where the sensing operation $\mathcal{S}$ is stored, $ER = [ER_{min}, ER_{max}]$
$METADATA$	Metadata memory region; contains ER_{min} and ER_{max}
ATok	Fixed memory region from which Verify reads the authorization token when called
GPIO	Memory region that is mapped to GPIO port
VR	Memory region storing Verify code which instantiates $VRASED$ software and its hardware protection
i_{Auth}	A fixed address in ROM, only be reachable (i.e., $PC = \mathsf{i}_{Auth}$) by a successful Verify call (i.e., Verify returns $\top$)
eKR	(Optional) memory region for the encryption key $\mathcal{K}_{enc}$ necessary to encrypt the $\mathcal{S}$ output (relevant to sensed data)

properties, we introduce the following two macros:

$$Read\_Mem(i) \equiv (R_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$$

$$Write\_Mem(i) \equiv (W_{en} \wedge D_{addr} = i) \vee (DMA_{en} \wedge DMA_{addr} = i)$$

representing read/write from/to a particular memory address i by either CPU or DMA. For reads/writes from/to some continuous memory region (composed of multiple addresses) $\mathcal{M} = [\mathcal{M}_{min}, \mathcal{M}_{max}]$, we instead say $D_{addr} \in \mathcal{M}$ to denote that $D_{addr} = i \wedge (i \geq \mathcal{M}_{min}) \wedge (i \leq \mathcal{M}_{max})$. The same holds for notation $DMA_{addr} \in \mathcal{M}$.

5.6.1 *VERSA*: Construction

Recall the key features of *VERSA* from Section 5.3. To guarantee *Mandatory Sensing Operation Authorization* and *Atomic Sensing Operation Execution*, *VERSA* constructs $PfB = (\mathsf{Authorize}, \mathsf{Verify}, \mathsf{XSensing})$ algorithms as in Construction 5.1. We describe each algorithm below.

5.6.1.1 Authorize

To authorize $\mathcal{S}$, $\mathcal{C}\mathsf{trl}$ picks a monotonically increasing $\mathcal{C}\mathsf{hal}$ and generates $\mathsf{ATok} := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), \mathcal{S})$. This follows *VRASED* authentication algorithm – see *VERSA* Verify specification below. ATok is computed over $\mathcal{S}$ with a one-time key derived from $\mathcal{K}$ and $\mathcal{C}\mathsf{hal}$, where $\mathcal{K}$ is the master secret key shared between $\mathcal{C}\mathsf{trl}$ and $\mathcal{D}\mathsf{ev}$.

5.6.1.2 Verify

To securely verify that an executable $\mathcal{S}$', installed in ER, matches authorized $\mathcal{S}$, $\mathcal{D}\mathsf{ev}$ invokes *VRASED*[*] to compute: $\sigma := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), \mathcal{S}')$ Verify outputs $\top$, if and only if $\sigma = \mathsf{ATok}$. In this case, PC reaches a fixed address, called i_{Auth}. Otherwise, Verify outputs $\perp$. In the rest of this section, we use "authorized software" to refer to software located in ER, for which $\mathsf{Verify}(ER, \mathsf{ATok})$ outputs $\top$. Whereas, "unauthorized software" refers to any software for which $\mathsf{Verify}(ER, \mathsf{ATok})$ outputs $\perp$.

[*] $\mathcal{D}\mathsf{ev}$ and $\mathcal{C}\mathsf{trl}$ act as $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$ in *VRASED* respectively.

CONSTRUCTION 5.1. *Let $\mathcal{K}$ is a symmetric key pre-shared between $\mathcal{C}$trl and VRASED in $\mathcal{D}$ev. VERSA instantiates a PfB =* [Authorize, Verify, XSensing] *scheme as follows:*

1. Authorize$^{\mathcal{C}\mathsf{trl}}(\mathcal{S})$*: $\mathcal{C}$trl produces an authorization message $\mathsf{M} := (\mathcal{S}, \mathcal{C}\mathsf{hal}, \mathsf{ATok})$, where $\mathcal{S}$ is a software, i.e., a sequence of instructions $\{\mathsf{i}_1, ..., \mathsf{i}_n\}$, that $\mathcal{C}$trl wants to execute on $\mathcal{D}$ev; $\mathcal{C}$hal is a monotonically increasing challenge; and ATok is an authentication token computed as shown in equation 5.7 . $\mathcal{C}$trl sends M to $\mathcal{D}$ev. Upon receiving M, $\mathcal{D}$ev is expected to parse M, find the memory region for $\mathcal{S}$, and execute* Verify *(see below).*

$$\mathsf{ATok} := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), \mathcal{S}) \tag{5.7}$$

2. Verify$^{\mathcal{D}\mathsf{ev}}(ER, \mathsf{ATok}, \mathcal{C}\mathsf{hal})$*: calls VRASED [47] on memory region $ER := [ER_{min}, ER_{max}]$ to securely compute σ as shown in equation 5.8.*

$$\sigma := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), ER) \tag{5.8}$$

If $\sigma = \mathsf{ATok}$, output $\top$; Otherwise, output $\bot$.

3. XSensing$^{\mathcal{D}\mathsf{ev}}(ER)$*: starts execution of software in ER by jumping to ER_{min} (i.e., setting $PC = ER_{min}$). A benign call to* XSensing *with input ER is expected to occur after one successful computation of* Verify *for the same ER region. Otherwise, VERSA hardware (see Construction 5.2) will cause the MCU to reset when GPIO is read.* XSensing *produces $\mathsf{E} := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$, the set of states produced by executing ER, and outputs $\top$ or $\bot$ as follows:*

$$\mathsf{XSensing}(ER) = \begin{cases} (\mathsf{E}, \top), & \text{if } \exists \mathsf{s} \in \mathsf{E} \text{ such that } (\mathsf{s} \in READ_{\mathsf{GPIO}}) \wedge (\mathsf{s} \notin RESET) \\ (\mathsf{E}, \bot), & \text{otherwise} \end{cases} \tag{5.9}$$

Figure 5.9: Verified Remote Sensing Authorization (*VERSA*) Scheme

Construction 5.2. *VERSA* HardwareMonitor
At all times, VERSA verified hardware enforces all following LTL properties :

A – Read-Access Control to *GPIO*:

$$\mathbf{G}: \{(Read\_Mem(\textit{GPIO}) \wedge \neg(PC \in ER)) \rightarrow \mathbf{Reset}\} \tag{5.10}$$

$$\mathbf{G}: \{[(PC = ER_{max}) \vee \mathbf{Reset}] \rightarrow (\neg Read\_Mem(\textit{GPIO}) \vee \mathbf{Reset}) \ \boldsymbol{W} \ (PC = \mathsf{i}_{Auth})\} \tag{5.11}$$

B – Ephemeral Immutability of ER and *METADATA*

$$\mathbf{G}: \{(PC = \mathsf{i}_{Auth}) \wedge (Write\_Mem(ER) \vee Write\_Mem(METADATA)) \rightarrow \mathbf{Reset}\} \tag{5.12}$$

$$\mathbf{G}: \{((Write\_Mem(ER) \vee Write\_Mem(METADATA) \rightarrow (\neg Read\_Mem(\textit{GPIO}) \vee \mathbf{Reset}) \ \boldsymbol{W} \ (PC = \mathsf{i}_{Auth}))\} \tag{5.13}$$

$$\text{[Optional] } \mathbf{G}: \{((Write\_Mem(ER) \vee Write\_Mem(METADATA) \rightarrow (\neg Read\_Mem(eKR) \vee \mathbf{Reset}) \ \boldsymbol{W} \ (PC = \mathsf{i}_{Auth}))\} \tag{5.14}$$

C – Atomicity and Controlled Invocation of ER:

$$\mathbf{G}: \{\neg\mathbf{Reset} \wedge (PC \in ER) \wedge \neg\mathbf{X}(PC \in ER) \rightarrow (PC = ER_{max}) \vee \mathbf{X}(\mathbf{Reset})\} \tag{5.15}$$

$$\mathbf{G}: \{\neg\mathbf{Reset} \wedge \neg(PC \in ER) \wedge \mathbf{X}(PC \in ER) \rightarrow \mathbf{X}(PC = ER_{min}) \vee \mathbf{X}(\mathbf{Reset})\} \tag{5.16}$$

$$\mathbf{G}: \{(PC \in ER) \wedge (irq \vee DMA_{en}) \rightarrow \mathbf{Reset}\} \tag{5.17}$$

[Optional] Read/Write-Access Control to Encryption Key ($\mathcal{K}_{enc}$) in eKR:

$$\mathbf{G}: \{(Read\_Mem(eKR) \wedge \neg(PC \in ER)) \rightarrow \mathbf{Reset}\} \tag{5.18}$$

$$\mathbf{G}: \{[(PC = ER_{max}) \vee \mathbf{Reset}] \rightarrow (\neg Read\_Mem(eKR) \vee \mathbf{Reset}) \ \boldsymbol{W} \ (PC = \mathsf{i}_{Auth})\} \tag{5.19}$$

$$\mathbf{G}: \{[Write\_Mem(eKR) \wedge \neg(PC \in VR)] \rightarrow \mathbf{Reset}\} \tag{5.20}$$

Remark: *[Optional] properties are needed only if support for encryption of outputs is desired.*

Figure 5.10: *VERSA* HardwareMonitor Specifications

5.6.1.3 XSensing

When XSensing (ER) is invoked, PC jumps to ER_{min}, and starts executing the code in ER. It produces a set $\boldsymbol{E}$ of states by executing ER, and outputs $\top$, if there is at least one state that reads GPIO without triggering an MCU reset. Otherwise, it outputs $\bot$.

5.6.1.4 HardwareMonitor

VERSA HardwareMonitor is verified to enforce LTL specifications (5.10)–(5.20) in Construction 5.1.

A – Read-Access Control to GPIO is jointly specified by LTLs (5.10) and (5.11). LTL (5.10) states that GPIO can only be read during execution of ER ($PC \in ER$), requiring an MCU reset otherwise. LTL (5.11) forbids all GPIO reads (even those within ER execution) before successful computation of Verify on ER binary using a valid ATok. Successful Verify computation is captured by condition $PC = \mathsf{i}_{Auth}$. A new successful computation of Verify(ER, ATok) is necessary whenever ER execution completes ($PC = ER_{max}$) or after reset/boot. Hence, each legitimate ATok can be used to authorize ER execution once.

B – Ephemeral Immutability of ER and *METADATA* is specified by LTLs (5.12)-(5.14). From the time when ER binary is authorized until it starts executing, no modifications to ER or *METADATA* are allowed. LTL (5.12) specifies that no such modification is allowed at the moment when verification succeeds ($PC = \mathsf{i}_{Auth}$); LTL (5.13) requires ER to be re-authorized from scratch if ER or *METADATA* are ever modified. Whenever these modifications are detected ($Write\_Mem(ER) \vee Write\_Mem(METADATA)$) further reads to GPIO are immediately blocked ($\neg Read\_Mem(\texttt{GPIO}) \vee \mathbf{Reset}$) until subsequent re-authorization of ER is completed (... $\mathbf{W}$ ($PC = \mathsf{i}_{Auth}$)). LTL (5.14) specifies the same requirement in order to read *VERSA*-provided encryption key ($\mathcal{K}_{enc}$) which is stored in mem-

ory region eKR. This property is only required when support for encryption of outputs is desired.

C – Atomicity & Controlled Invocation of ER are enforced by LTLs (5.15), (5.16), and (5.17). They specify that ER execution must start at ER_{min} and end at ER_{max}. Specifically, they use the relation between *current* and *next* PC values. The only legal PC transition from currently outside of ER to next inside ER is via $PC = ER_{min}$. Similarly, the only legal PC transition from currently inside ER to next outside ER is via $PC = ER_{max}$. All other cases trigger an MCU reset. In addition, LTL (5.17) requires an MCU reset whenever interrupts or DMA activity is detected during ER execution. This is done by simply checking irq and DMA_{en} signals.

We note that XSensing relies on the HardwareMonitor to reset the MCU when a violation of ER atomic execution is detected. Upon reset all data is erased. However, when execution of $\mathcal{S}$ completes successfully *VERSA* does not trigger resets. In this case, $\mathcal{S}$ is responsible for erasing its own stack before completion. We discuss how this self-clean-up routine can be implemented in Section 5.10.1.

5.6.2 Encryption & Integrity of ER Output

Recall that, $\mathcal{S}$ might need to encrypt and send the result to $\mathcal{C}$trl. For that purpose, Verify derives a fresh one-time encryption key ($\mathcal{K}_{enc}$) from $\mathcal{K}$ and $\mathcal{C}$hal. To assure confidentiality of $\mathcal{K}_{enc}$, the following properties are required for the memory region (eKR) reserved to store $\mathcal{K}_{enc}$:

1. eKR is writable only by Verify (i.e., $PC \in VR$); and
2. eKR is readable only by ER after authorization.

LTLs (5.18)-(5.20) and (5.14) specify the confidentiality requirements of $\mathcal{K}_{enc}$. In sum, these properties establish the same read access-control policy for eKR and `GPIO` regions. Therefore, only authorized $\mathcal{S}$ is able to retrieve $\mathcal{K}_{enc}$.

5.7 Verified Implementation & Security Analysis

This section describes implementation details of *VERSA*. We also describe how *VERSA* hardware is formally verified.

5.7.1 Sub-module Implementation & Verification

VERSA sub-modules are represented as FSMs and individually verified to hold for LTL properties from Construction 5.2. They are implemented in Verilog HDL as Mealy machines, i.e., their output is determined by both their current state and current inputs. Each FSM has a single output: a local **Reset**. *VERSA* global output **Reset** is given by the disjunction (logic OR) of all local **Reset**-s. For simplicity, instead of explicitly representing the output **Reset** value for each state, we use the following convention:

1. **Reset** is 1 whenever an FSM transitions to $RESET$ state;
2. **Reset** remains 1 while on $RESET$ state;
3. **Reset** is 0 otherwise.

Note that all FSMs remain in $RESET$ state until $PC = 0$ which indicates that the MCU reset routine finished.

Fig. 5.11 illustrates the *VERSA* sub-module that implements read-access control to `GPIO` and eKR (when applicable). It guarantees that such reads are only possible when they

emanate from execution of authorized software $\mathcal{S}$ contained in ER. It also assures that no modifications to ER or $METADATA$ occur between authorization of $\mathcal{S}$ and its subsequent execution. The Verilog implementation of this FSM is formally verified to adhere to LTLs (5.10)-(5.14) and (5.18)-(5.19). It has 3 states: (1) $rLOCK$, when reads to `GPIO` (and possibly eKR) are disallowed; (2) $rUNLOCK$, when such reads are allowed to ER; and (3) $RESET$. The initial state (after reset or boot) is $RESET$, and it switches to $rLOCK$ state when $PC = 0$. It switches to $rUNLOCK$ when $PC = \mathsf{i}_{Auth}$ (with no reads to `GPIO` and eKR), indicating that Verify was successful. Note that $rUNLOCK$ transitions to $RESET$ when reads are attempted from outside ER, thus preventing reads by any unauthorized software. Once PC reaches ER_{max}, indicating that ER execution has finished, the FSM transitions back to $rLOCK$. Also, any attempted modifications to $METADATA$ or ER in $rUNLOCK$ state bring the FSM back to $rLOCK$. Note that $rUNLOCK$ is only reachable after authorization of ER, i.e., $PC = \mathsf{i}_{Auth}$.

The FSM in Figure 5.12 enforces LTL (5.20) to protect eKR from external writes. It has two states: (1) $wUNLOCK$, when writes to eKR are allowed; and (2) $RESET$. At boot/after reset ($PC = 0$), this FSM transitions from $RESET$ to $wUNLOCK$. It transitions back to $RESET$ state whenever writes to eKR are attempted, unless these writes come from Verify execution ($PC \in VR$).

Figure 5.13 shows the FSM verified to enforce ER atomicity and controlled invocation: LTLs (5.15)-(5.17). It has five states; $notER$ and $midER$ correspond to PC being outside and within ER (not including ER_{min} and ER_{max}), respectively. $firstER$ and $lastER$ are states in which PC points to ER_{min} and ER_{max}, respectively. The only path from $notER$ to $midER$ is via $firstER$. Likewise, the only path from $midER$ to $notER$ is via $lastER$. The FSM transitions to $RESET$ whenever PC transitions do not follow aforementioned paths. It also transitions to $RESET$ (from any state other than $notER$) if irq or DMA_{en} signals are set.

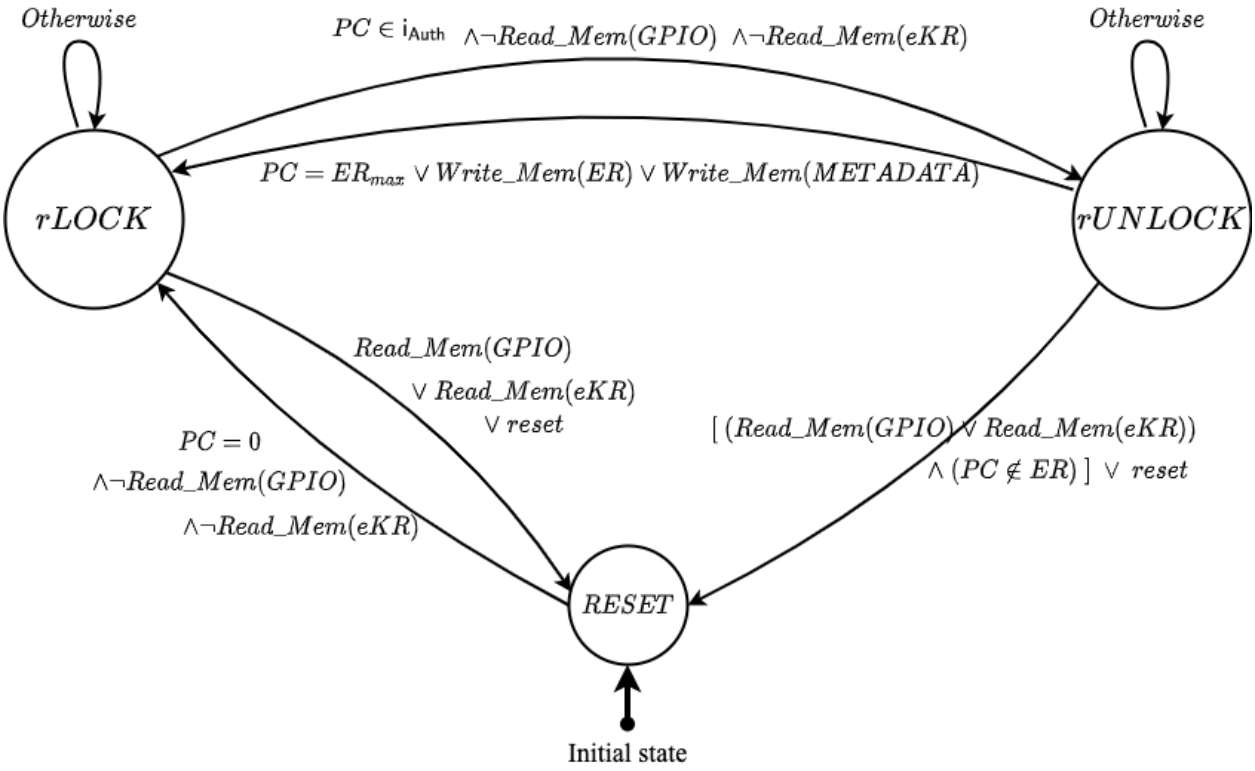

Figure 5.11: Verified FSM for `GPIO` and eKR Read-Access Control (LTL (5.10)-(5.14) & LTL (5.18)-(5.19))

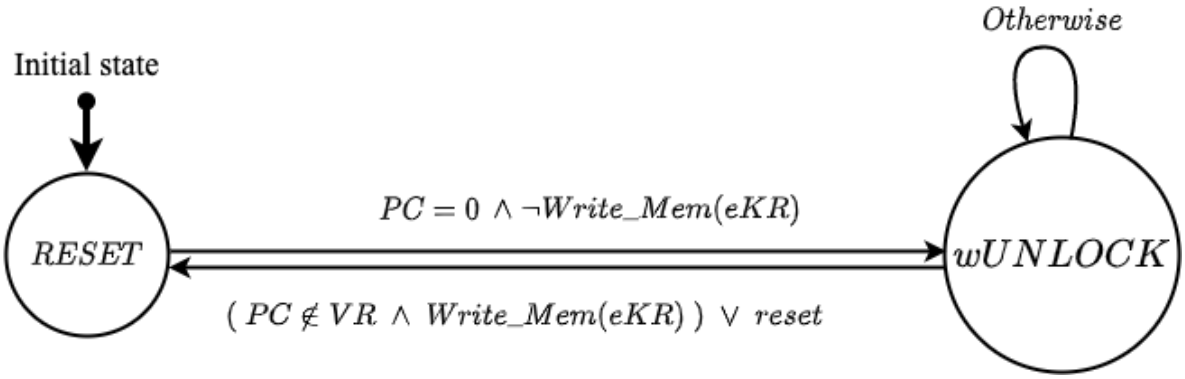

Figure 5.12: Verified FSM for eKR Write-Access Control (LTL (5.20))

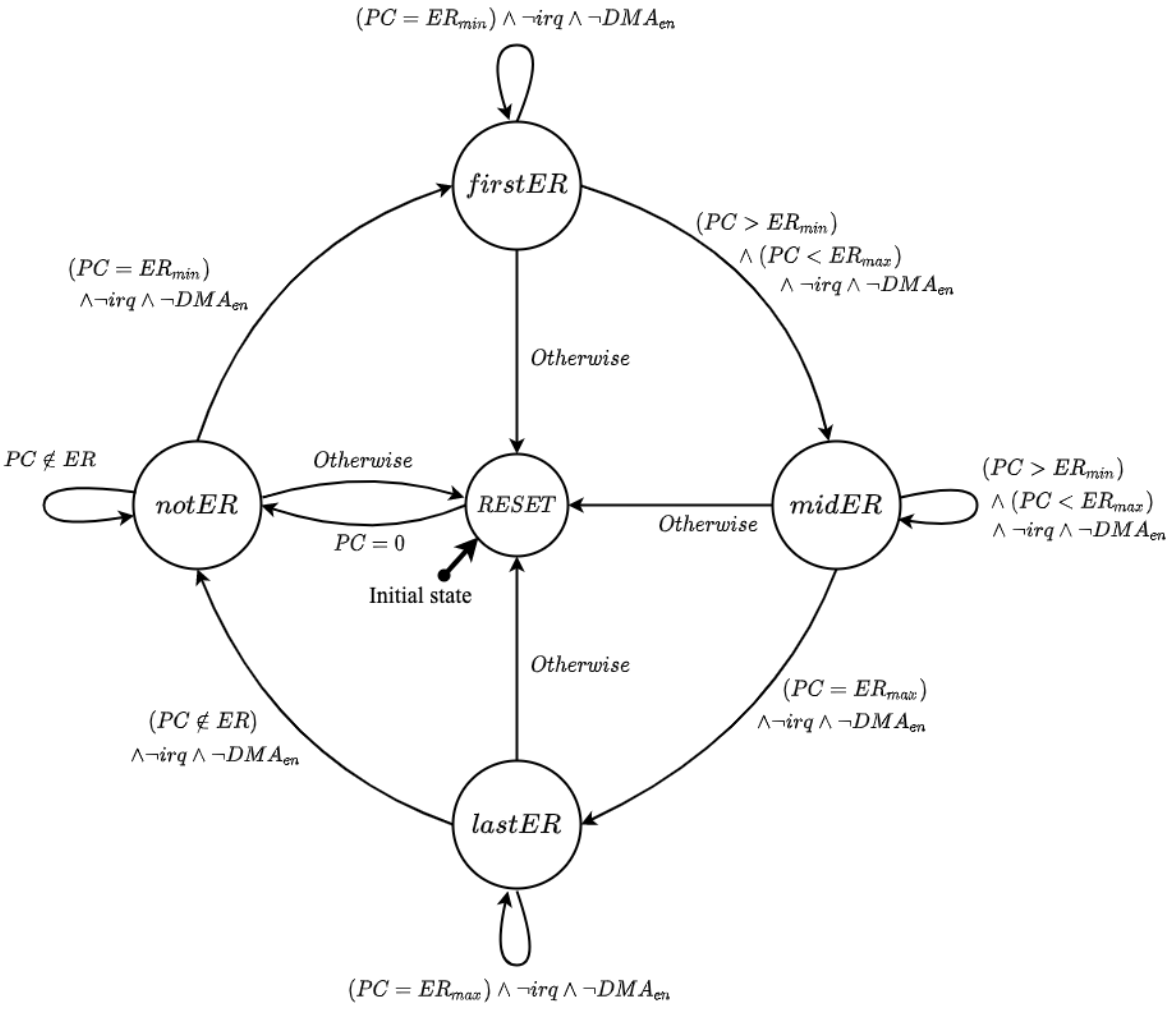

Figure 5.13: ER Atomicity and Controlled Invocation FSM (LTL (5.15)-(5.17))

Definition 5.5. *Atomic Sensing Operation Execution:*

$$\begin{aligned} & \boldsymbol{G}\{\ (PC \in ER) \rightarrow [(PC \in ER) \wedge \neg irq \wedge \neg DMA_{en}]\ \boldsymbol{W}\ [(PC = ER_{max}) \vee \textbf{Reset}]\ \} \\ \wedge \quad & \boldsymbol{G}\{\ \neg\textbf{Reset} \wedge \neg(PC \in ER) \wedge \boldsymbol{X}(PC \in ER) \rightarrow \boldsymbol{X}(PC = ER_{min}) \vee \boldsymbol{X}(\textbf{Reset})\ \} \end{aligned} \tag{5.21}$$

Definition 5.6. *Mandatory Sensing Operation Authorization:*

$$\begin{aligned} & \boldsymbol{G}\{\ (Read\_Mem(\texttt{GPIO}) \wedge \neg\textbf{Reset}) \rightarrow (PC \in ER)\ \} \wedge \\ & \Big\{(PC = \mathsf{i}_{Auth}) \wedge \big\{\quad (PC = \mathsf{i}_{Auth}) \rightarrow [\neg Write\_Mem(ER) \wedge \neg Write\_Mem(METADATA) \\ & \qquad \wedge (Write\_Mem(eKR) \rightarrow (PC \in VR))]\ \boldsymbol{U}\ (PC = ER_{min})\big\} \\ & \Big\}\ \boldsymbol{B}\ \big\{Read\_Mem(\texttt{GPIO}) \wedge \neg\textbf{Reset}\big\} \end{aligned} \tag{5.22}$$

Figure 5.14: *VERSA* End-To-End Security Properties in LTL.

5.7.2 Sub-module Composition and *VERSA* End-To-End Security

To demonstrate security of *VERSA* according to Definition 5.4, our strategy is two-pronged:

A) We show that LTL properties from Construction 5.1 are sufficient to imply that GPIO (and eKR) is only readable by $\mathcal{S}$ and any XSensing operation that returns $\top$ (i.e., performs sensing) is executed atomically. The former is formally specified in Definition 5.6, and the latter in Definition 5.5. For this part, we write an LTL computer proof using SPOT LTL proof assistant [60].

B) We use a cryptographic reduction to show that, as long as item **A** holds, *VRASED* security can be reduced to *VERSA* security according to Definition 5.4.

The intuition for this strategy is that, to win *PfB*-game in Definition 5.4, $\mathcal{A}$dv must either break the atomicity of XSensing (which is in direct conflict with Definition 5.5) or execute XSensing with unauthorized software and read GPIO without causing an MCU **Reset**. Definition 5.6 guarantees that the latter is not possible without a prior successful call to Verify. On the other hand, Verify is implemented using *VRASED* verified architecture, which guar-

Theorem 5.1. *Definition 5.2 ∧ LTL 5.15, 5.16, 5.17 → Definition 5.5*

Theorem 5.2. *Definition 5.2 ∧ LTL 5.10, 5.11, 5.12, 5.13, 5.16, 5.20 → Definition 5.6*

Theorem 5.3. *VERSA is secure according to the PfB-game in Definition 5.4, as long as VRASED is a secure* $\mathcal{RA}$ *architecture according to VRASED security game from [47].*

Figure 5.15: *VERSA* Theorems for proving end-to-end security.

antees the unforgeability of ATok. Hence, breaking *VERSA* requires either violating *VERSA* verified guarantees or breaking *VRASED* verified guarantees, which should be infeasible to any PPT $\mathcal{A}\mathsf{dv}$.

Section 5.8 includes proofs for Theorems 5.1, 5.2, and 5.3, in accordance to this proof strategy. The rest of this section focuses on *VERSA* end-to-end implementation goals captured by LTLs in Definitions 5.5 and 5.6 as well as their relation to *VERSA* high-level features discussed in Section 5.3.

[Definition 5.5] states that it **globally** (always) holds that ER is atomically executed with controlled invocation. That is, whenever an instruction in ER executes ($PC \in ER$), it keeps executing instructions within ER ($PC \in ER$), with no interrupts and no DMA enabled, **until** PC reaches the last instruction in ER_{max} or an MCU reset occurs. Also, if an instruction in ER starts to execute, it always begins with the first instruction in ER_{min}. This formally specifies the *Atomic Sensing Operation Execution* feature discussed in Section 5.3.

[Definition 5.6] **globally** requires that whenever GPIO is successfully read (i.e., without a **Reset**), this read must come from the CPU while ER is being executed. In addition, **before** this read operation, the following must have happened at least once:

(1) Verify succeeded (i.e., $PC = \mathsf{i}_{Auth}$);

(2) From the time when $PC = \mathsf{i}_{Auth}$ **until** ER starts executing (i.e., $PC = ER_{min}$), no

modification to ER and $METADATA$ occurred; and

(3) If there was any write to eKR from the time when $PC = \mathsf{i}_{Auth}$, **until** $PC = ER_{min}$, it must have been from Verify, i.e., while $PC \in VR$.

This formally specifies the intended behavior of the *Mandatory Sensing Operation Authorization* feature, discussed in Section 5.3.

5.8 *VERSA* Composition Proof

In this section, we show that *VERSA* is a secure *PfB* architecture according to Definition 5.4, as long as **A)** the sub-properties in Construction 5.1 hold (Theorem 5.1, 5.2) and **B)** *VRASED* is a secure remote attestation ($\mathcal{RA}$) architecture according to the *VRASED* security definition in [47] (Theorem 5.3). Informally, **(A)** shows that if the machine model and all LTLs in Construction 5.1 hold, then the end-to-end goals for secure *PfB* architecture are met, while this does not include the goal of prevention of forging authorization tokens. **(B)** handles the latter using a cryptographic reduction, i.e., it shows that an adversary able to forge the authorization token (with more than negligible probability) can also break *VRASED* according to the $\mathcal{RA}$-game, which is a contradiction assuming the security of *VRASED*. Therefore, Theorems 5.1-5.3 prove that *VERSA* is a secure *PfB* architecture as long as *VRASED* is a secure $\mathcal{RA}$ architecture.

For **(A)**, computer-checked LTL proofs are performed using SPOT LTL proof assistant [60]. These proofs are available at [12]. We present the intuition behind them below.

Proof of Theorem 5.1 (Intuition). LTL (5.16) states the legal entry instruction requirement, while LTL (5.15) states the legal exit instruction requirement in atomicExec. Also, since LTL

(5.15) states that ER_{max} is the only exit from ER without a reset, it implies self-contained execution of ER. Lastly, LTL (5.17) enforces MCU reset if any interrupt or DMA occurs, which prevents interrupts and DMA actions, as required by atomicExec. These imply the LTL in Definition 5.5 which stipulates that execution of ER must start with ER_{min} and stays within ER with no interrupts nor DMA actions until PC reaches ER_{max} (causing a reset otherwise). □

Proof of Theorem 5.2 (Intuition). Definition 5.6 **(i)** requires at least one successful verification of ER before GPIO can be read successfully (without triggering a reset); and **(ii)** disallows modifications to ER, *METADATA*, and $\mathcal{K}_{enc}$ (other than by VR) in between ER verification subsequent ER execution. LTLs (5.10) and (5.11) state that PC must be within ER to read GPIO and disallow GPIO reads by default (including when MCU reset occurs) and after the execution of ER is over ($PC = ER_{max}$). Also, LTL (5.11) requires (re-)authorization ($PC = \mathsf{i}_{Auth}$) of ER after the execution of ER is over ($PC = ER_{max}$). LTL (5.13) disallows GPIO reads until the (re-)verification whenever ER or *METADATA* are written. LTL (5.12) disallows changes to ER and *METADATA* at the exact time when verification succeeds. LTL (5.16) guarantees that the execution of ER starts with ER_{min} and LTL (5.20) guarantees that only the VR code can modify the value in eKR. Thus, these are sufficient imply Definition 5.6. □

For **(B)**, we construct a reduction from the security game of *VRASED* in [47] to the security game of *VERSA* according to the Definition 5.4. i.e., the ability to break the *PfB*-game of *VERSA* allows breaking the $\mathcal{R}$A-game of *VRASED*. Therefore, as long as *VRASED* is a secure $\mathcal{R}$A architecture according to the $\mathcal{R}$A-game, *VERSA* is secure according to the *PfB*-game.

Proof of Theorem 5.3. Assume $\mathcal{A}\mathsf{dv}_{PfB}$, an adversary who can win the security game in Definition 5.4 against *VERSA* with more than negligible probability. We show that if such $\mathcal{A}\mathsf{dv}_{PfB}$ exists, then it can be used to construct $\mathcal{A}\mathsf{dv}_{\mathcal{RA}}$ that wins the $\mathcal{RA}$-game with more than negligible probability.

Recall that, to win the *PfB*-game, $\mathcal{A}\mathsf{dv}_{PfB}$ must trigger $\top$ as a result of $\mathsf{XSensing}$, which means it reads the sensed data without MCU reset. From the *PfB*-game step 4 in Definition 5.4, it can be done in either of the following two ways:

`Case1.` $\mathcal{A}\mathsf{dv}_{PfB}$ executes a new, unauthorized software $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$ which causes $\mathsf{XSensing}(\mathcal{S}_{\mathcal{A}\mathsf{dv}}) \rightarrow \top$; or

`Case2.` $\mathcal{A}\mathsf{dv}_{PfB}$ breaks the atomic execution of an authorized, but not yet executed software, $\mathcal{S}_j$, so that it causes $\mathsf{XSensing}(\mathcal{S}_j) \rightarrow (E, \top)$ such that $\mathsf{atomicExec}(E, \mathcal{S}_j) \equiv \perp$.

Recall that for the instruction set I_j of $\mathcal{S}_j$ and a set E_j of execution states , to have $\mathsf{atomicExec}(I_j, E_j) \equiv \perp$, at least one of four requirements in Definition 5.4.1 must be false. Note that the atomic sensing operation execution goal in Definition 5.5 rules out `Case2`. Specifically, LTL (5.16) enforces (1), while (2) and (3) are guaranteed by LTL (5.15). Lastly, (4) is covered by LTL (5.17).

For `Case1`, $\mathcal{A}\mathsf{dv}_{PfB}$ needs to read `GPIO` without causing an MCU reset. Recall that the *Mandatory Sensing Operation Authorization* in Definition 5.6 requires Verify (with input executable in ER) to succeed at least once before reading `GPIO`. According to *VERSA* construction, $\mathcal{A}\mathsf{dv}_{PfB}$ causes $\mathsf{Verify}(ER, \mathsf{ATok}^*, \mathcal{C}\mathsf{hal}^*)$ to output $\top$, where ER contains $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$ which is an unauthorized software, ATok^* is a valid issued (but never used) token, and $\mathcal{C}\mathsf{hal}^*$ is its corresponding challenge. Since Verify is implemented using *VRASED* to compute HMAC of $\mathcal{C}\mathsf{hal}$ and ER, $\mathcal{A}\mathsf{dv}_{PfB}$ can be directly used as $\mathcal{A}\mathsf{dv}_{\mathcal{RA}}$ to win the $\mathcal{RA}$-game of *VRASED*. Thus, assuming secure *VRASED*, this is a contradiction, which implies the security of *VERSA* according to the *PfB*-game. □

5.9 Evaluation

In this section, we discuss *VERSA* implementation details and evaluation. *VERSA* source code and verification/proofs are publicly available at [12].

5.9.1 Toolchain & Prototype Details

VERSA is built atop OpenMSP430 [68]: an open source implementation of TI-MSP430 [74]. We use Xilinx Vivado to synthesize an RTL description of HardwareMonitor and deploy it on Diligent Basys3 prototyping board for Artix7 FPGA. For the software part (mostly to implement Verify), *VERSA* extends *VRASED* software (which computes $HMAC$ over $\mathcal{D}$ev memory) to include a comparison with the received ATok (See Section 5.9.4 for extension details). The comparison uses secure `memcmp` (constant-time function) operation to check whether σ (from equation 5.8) computed over ER, matches ATok. It also supports to write $\mathcal{K}_{enc}$ generated during Verify to eKR using `memcpy`.

We use the NuSMV model checker to formally verify that HardwareMonitor implementation adheres to LTL specifications (5.10)-(5.20).

Table 5.2: Hardware Overhead & Verification cost

Architecture	Hardware		Reserved	Verification			
	LUTs	Regs	RAM (bytes)	LoC	#(LTLs)	Time (s)	RAM (MB)
OpenMSP430	1854	692	0	-	-	-	-
VRASED	1891	724	2332	481	10	0.4	13.6
VERSA + *VRASED*	2109	742	2336	1118	21	13956.4	1059.1

5.9.2 Hardware Overhead

Table 5.2 reports on *VERSA* hardware overhead, as compared to unmodified OpenMSP430 and *VRASED*. Similar to other schemes [47, 49, 103, 62], we consider hardware overhead

in terms of additional Look-Up Tables (LUTs) and registers. Extra hardware in terms of LUTs gives an estimate of additional chip cost and size required for combinatorial logic, while extra hardware in terms of registers gives an estimate of memory overhead required by sequential logic in *VERSA* FSMs. Compared to *VRASED*, *VERSA* requires 10% additional LUTs and 2% additional registers. In actual numbers, it adds 255 LUTs and 50 registers to the underlying MCU as shown in Table 5.2.

5.9.3 Verification Costs

Formal verification costs are reported in Table 5.2. We use a Ubuntu 18.04 desktop machine running at 3.4GHz with 32GB of RAM for formal verification. Our verification pipeline converts Verilog HDL to SMV specification language and then verifies it against the LTL properties listed in Construction 5.1 using the NuSMV model checker (per Section 5.2). *VERSA* verification requires checking 11 extra invariants – LTLs (5.10) to (5.20) – in addition to *VRASED* LTL invariants. It also incurs higher run-time and memory usage than *VRASED* verification. This is due to two additional 16-bit hardware signals (ER_{min}, ER_{max}) which increase the space of possible input combinations and thus the complexity of model checking process. However, verification is still manageable in a commodity desktop – it takes around 5 minutes and consumes 340MB of memory.

5.9.4 Runtime Overhead

VERSA requires any software piece seeking to access `GPIO` (and $\mathcal{K}_{enc}$) to be verified. Consequently, runtime overhead is due to Verify computation which instantiates *VRASED*. This runtime includes: **(1)** time to compute σ from equation 5.8; **(2)** time to check if $\sigma =$ `ATok`; and **(3)** time to write $\mathcal{K}_{enc}$ to eKR, when applicable. Naturally, the runtime overhead is dominated by the computation in **(1)** which is proportional to the size of ER. As *VERSA*

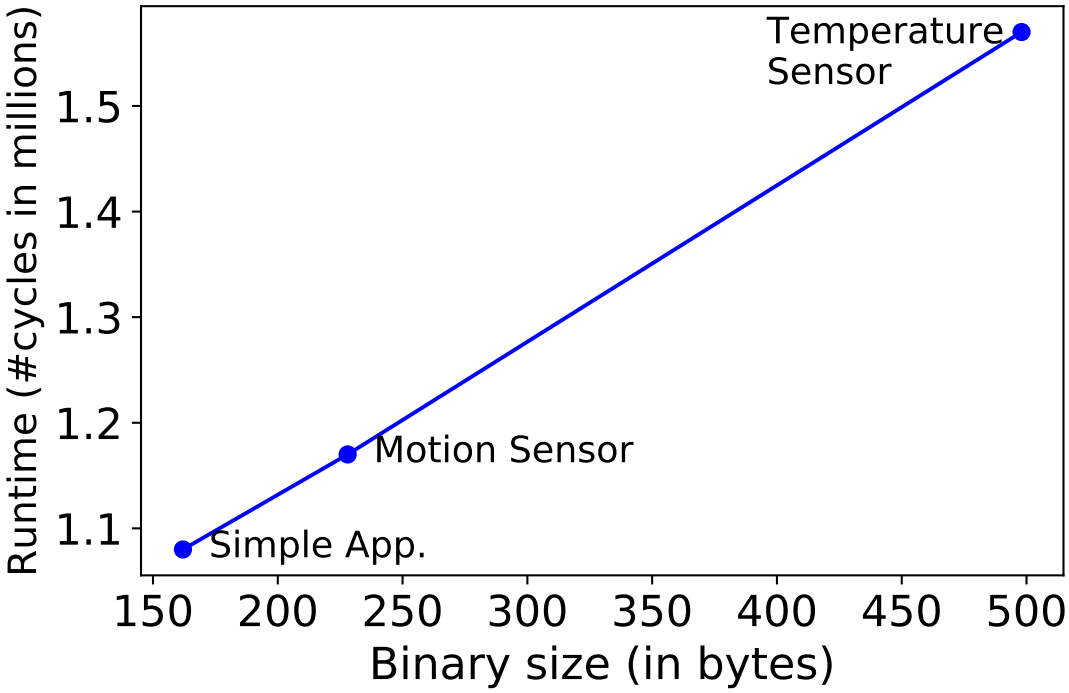

Figure 5.16: Runtime overhead of *VERSA* due to Verify

does not require any software instrumentation, there is no increase in code size or runtime overhead while executing ER.

We measure Verify cost on three sample applications: (1) Simple Application, which reads 32-bytes of `GPIO` input and encrypts it using One-Time-Pad (OTP) with $\mathcal{K}_{enc}$; (2) Motion Sensor (available at `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/pir_motion_sensor`) – which continuously reads `GPIO` input to detect movements and actuates a light source when movement is detected; and (3) Temperature Sensor (adapted code from `https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor` to support encryption of its outputs) – which reads ambient temperature via `GPIO` and encrypts this reading using OTP. We prototype using OTP for encryption for the sake of simplicity noting that *VERSA* does not mandate a particular encryption scheme (e.g., CPA/CCA-Secure ones). All sample applications include a self-clean-up code executed immediately before reaching their

exit point to erase their stack traces.

Figure 5.16 shows Verify runtimes on these applications. Assuming a clock frequency of 10MHz (a common frequency for low-end MCUs), Verify runtime ranges from $100 - 200$ milliseconds for these applications. The overhead is linear on the binary size.

Note: Every application includes a self-clean-up code at the end to erase its stack after execution. We present an example of such clean up code in Section 5.10.1.

Remark: *Runtime of* Verify *is almost the same as the runtime of attestation algorithm of other similar architectures for low-end MCUs such as [47, 62, 49] since they all use the same HMAC implementation from HACL* library. The additional runtime of* Verify *is due to* `memcmp` *and* `memcpy` *steps at the last, which is negligible compared to the HMAC computation step. Next, we do not compare runtime overhead of VERSA with SANCUS [103] as the latter is a pure hardware architecture with no software overhead.*

5.9.5 Comparison with Other Low-End Architectures:

To the best of our knowledge, *VERSA* is the first architecture related to *PfB*. However, to provide a point of reference in terms of performance and overhead, we compare *VERSA* with other low-end trusted hardware architectures, such as SMART [62], VRASED [47], APEX [49], and SANCUS [103]. All these architectures provide $\mathcal{R}$A-related services to attest integrity of software on $\mathcal{D}$ev either statically or at runtime. Since *PfB* also checks software integrity before granting access to `GPIO`, we consider these architectures to be related to *VERSA*.

Figures 5.17 and 5.18 compare *VERSA* hardware overhead with the aforementioned architectures in terms of additional LUTs and registers. Percentages are relative to the plain MSP430 core total cost.

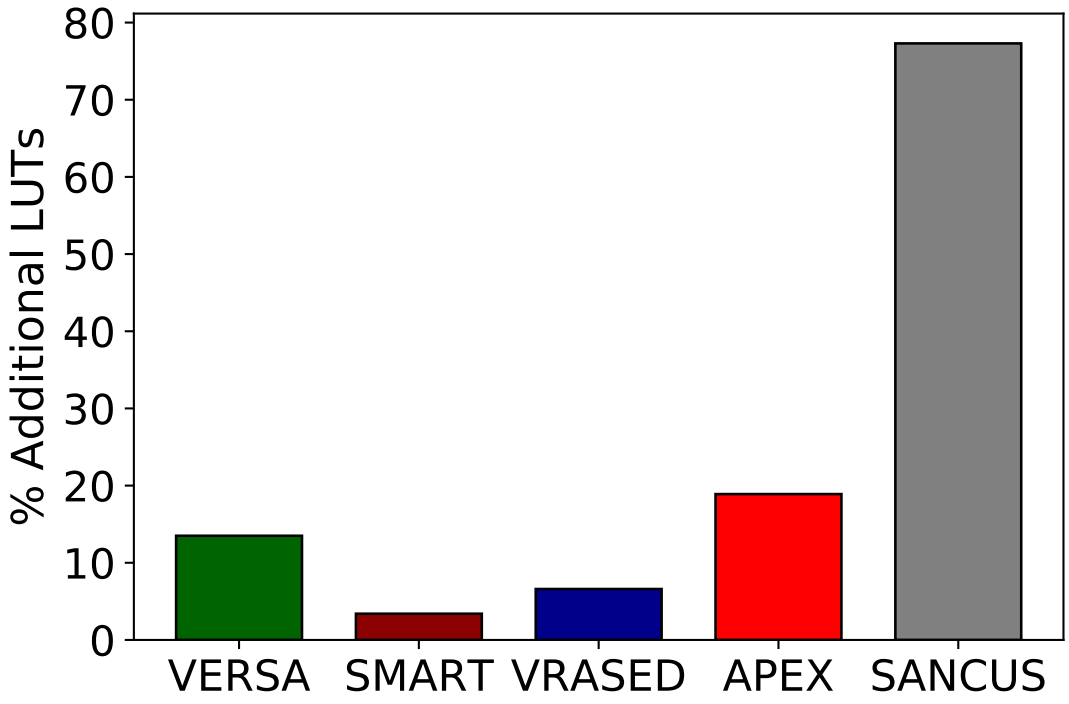

Figure 5.17: *VERSA* Additional HW overhead (%) in Number of Look-Up Tables

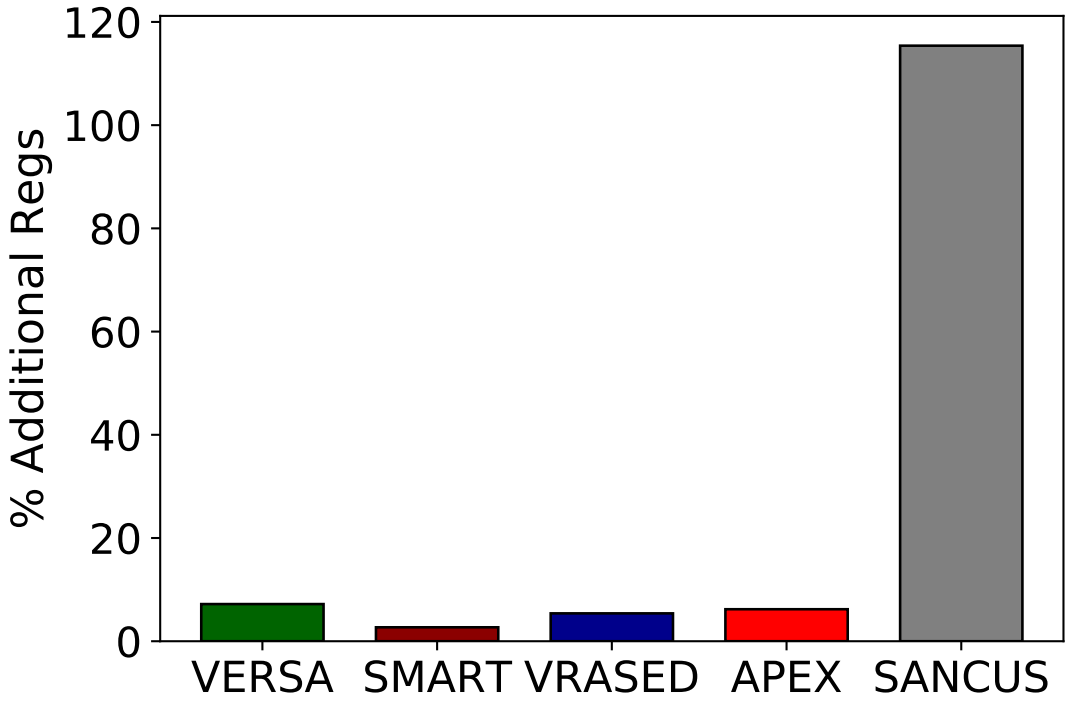

Figure 5.18: *VERSA* Additional HW overhead (%) in Number of Registers

VERSA builds on top of *VRASED*. As such, it is naturally more expensive than hybrid $\mathcal{RA}$ architectures such as SMART and *VRASED*. Similar to *VERSA*, APEX also monitors execution properties and also builds on top of $\mathcal{RA}$ (in APEX case, with the goal of producing proofs of remote software execution). Therefore, *VERSA* and APEX exhibit similar overheads. SANCUS presents a higher cost because it implements $\mathcal{RA}$ and isolation features in hardware – which makes it faster compared to hybrid schemes.

5.10 Discussion

5.10.1 Clean-up after Program Termination

While *VERSA* guarantees the confidentiality of sensing operations, it requires $\mathcal{S}$ to erase its own stack/heap before its termination. This ensures that unauthorized software can not extract and leak sensitive information from $\mathcal{S}$ execution and allocated data. This can be achieved via a single call to libc's `memset` function with starting address matching the base of $\mathcal{S}$ stack and size equal to the maximum size reached by $\mathcal{S}$ execution.

The maximum stack size can be determined manually by counting the allocated local variables in small and simple $\mathcal{S}$ implementations. To automatically determine this size in more complex $\mathcal{S}$ implementations, all functions called within $\mathcal{S}$ must update the highest point reached by their respective stacks. Figure 5.19 shows a sample application that reads 32 bytes of sensor data, encrypts this data using *VERSA* one-time key $\mathcal{K}_{enc}$, and cleans-up the stack thereafter. Line 12 is $\mathcal{S}$ entry point (ER_{min}). $\mathcal{S}$ first saves the stack pointer to `STACK_MIN` address. Then, the `application` function is called, in line 17. `application` implements $\mathcal{S}$ intended behavior. After the `application` is done, the clean up code (lines 51-53) is called with `STACK_MIN` as the start pointer and size of 32 + 4 bytes (32 bytes for `data` variable (in line 39) and 4 bytes for stack metadata).

```
1   #include <string.h>
2   // Sensor is at P3IN
3   #define P3IN          (*(volatile unsigned char *) 0x0018)
4   #define BIT4          0x10
5   #define HIGH          0x1
6   #define LOW           0x0
7   #define eKR           0x360
8   #define STACK_MIN     0x1010 // App stack start pointer is stored here
9   #define RESULT        0x1030 // App result is written here
10
11  // ER_min
12  __attribute__ ((section (".er.entry"), naked)) void applicationEntry() {
13      // Save the stack pointer to STACK_MIN at entry.
14      __asm__ volatile("mov    r1,   &0x1010" "\n\t");
15
16      // Call the application
17      application();
18
19      // Call the clean up code
20      cleanUp((uint8_t*)STACK_MIN, 32 + 4);
21
22      // Jump to applicationExit()
23      __asm__ volatile( "br #__er_leave" "\n\t");
24  }
25
26  __attribute__ ((section (".er.body"))) int digitalRead() {
27      if(P3IN & BIT4) return HIGH;
28      else return LOW;
29  }
30
31  __attribute__ ((section (".er.body"))) void encrypt(uint8_t* data, int
        data_size, uint8_t* key, int key_size) {
32      for (int i = 0; i < data_size; i++) {
33          data[i] = data[i] ^ key[i];
34      }
35  }
36
37  __attribute__ ((section (".er.body"))) void application() {
38      // Read sensor data
39      uint8_t data[32];
40      for (int i = 0; i < 32; i++) {
41          data[i] = digitalRead();
42      }
43
44      // One-Time-Pad encryption of data using the enc_key in eKR.
45      encrypt(data, 32, (uint8_t*)eKR, 32);
46
47      // Copy encrypted result to RESULT address.
48      memcpy(RESULT, data, 32);
49  }
50
51  __attribute__ ((section (".er.body"))) void cleanUp(uint8_t* start, int size){
52      memset((uint8_t*)start, 0, size);
53  }
54
55  //ER_max
56  __attribute__ ((section (".er.exit"), naked)) void applicationExit() {
57      __asm__ volatile("ret" "\n\t");
58  }
```

Figure 5.19: Sample sensing operation that reads GPIO input, encrypts it, and cleans up its stack after execution.

5.10.2 Data Erasure on Reset/Boot

Violations to *VERSA* properties trigger an MCU reset. A reset immediately stops execution and prepares the MCU core to reboot by clearing all registers and pointing the program counter (PC) to the first instruction of PMEM. However, some MCUs may not guarantee erasure of DMEM as a part of this process. Therefore, traces of data allocated by $\mathcal{S}$ (including sensor data) could persist across resets.

In MCUs that do not offer DMEM erasure on reset, a software-based Data Erasure (DE) can be implemented and invoked it as soon as the MCU starts, i.e., as a part of the bootloader code. In particular, DE can be implemented using `memset` (similar to lines 51-53) with constant arguments matching the entirety of the MCU's DMEM. DE should be immutable (e.g., stored in ROM) which is often the case for bootloader binaries. Upon reset, PC must always point to the first instruction of DE. The normal MCU start-up proceeds normally after DE execution is completed.

5.11 Limitations:

In the following, we discuss some limitations of *VERSA* and their possible mitigations.

5.11.1 Shared Libraries

To verify $\mathcal{S}$, $\mathcal{C}\mathsf{trl}$ must ensure that $\mathcal{S}$ spans one contiguous memory region (ER) on $\mathcal{D}\mathsf{ev}$. If any code dependencies exist outside of ER, *VERSA* resets the MCU according to LTL (5.17). To preclude this situation, $\mathcal{S}$ must be made self-contained by statically linking all of its dependencies within ER.

5.11.2 Atomic Execution & Interrupts

Per Definition 5.4, *VERSA* forbids interrupts during execution of XSensing. This can be problematic, especially on $\mathcal{D}$ev with strict real-time constraints. In this case, $\mathcal{D}$ev must be reset in order to allow servicing the interrupt after DMEM erasure. This can cause a delay that could be harmful to real-time settings. Trade-offs between privacy and real-time constraints should be carefully considered when using *VERSA*. One way to remedy this issue is to allow interrupts as long as all interrupt handlers are: (1) themselves immutable and uninterruptible from the start of XSensing until its end; and (2) included in ER memory range and are thus checked by Verify.

5.11.3 Possible Side-channel Attacks

MSP430 and similar MCU-s allow configuring some GPIO ports to trigger interrupts. If one of such ports is used for triggering an interrupt, $\mathcal{A}$dv could possibly look at the state of $\mathcal{D}$ev and learn information about GPIO data. For example, suppose that a button press mapped to a GPIO port triggers execution of a program that sends some fixed number of packets over the network. Then, $\mathcal{A}$dv can learn that the GPIO port was activated by observing network traffic. To prevent such attacks, privacy-sensitive quantities should always be physically connected to GPIO ports that do not interrupt sources (these are usually the vast majority of available GPIO ports). Other popular timing attacks related to cache side-channels and speculative execution, are not applicable to this class of devices, as these features are not present in low-end MCUs.

5.11.4 Flash Wear-Out

VERSA implements Verify using *VRASED*. As discussed in Section 2.2, the authentication protocol in *VRASED* requires persistent storage of the highest value of a monotonically increasing challenge/counter in flash. Flash memory has a limited number of write cycles (typically at least 10,000 cycles [10, 5]). Hence, a large number of successive counter updates may wear out that portion of flash memory, the persistent counter stored in flash is only updated following successful authentication of $\mathcal{C}$trl. Therefore, only legitimate requests from $\mathcal{C}$trl cause these flash writes. Nonetheless, if the number of expected legitimate calls to Verify is high, one must select the persistent storage type or (alternatively) use different flash blocks once a given flash block storing the counter reaches its write cycles' limit. For a more comprehensive discussion of this matter, see [25].

5.11.5 *VERSA* Alternative Use-Case

VERSA can be viewed as a general technique to control access to memory regions based on software authorization tokens. We apply this framework to GPIO in low-end MCUs. Other use-cases are possible. For example, a *VERSA*-like architecture could be used to mark a secure storage region and grant access only to explicitly authorized software. This could be useful if $\mathcal{D}$ev runs multiple (mutually distrusted) applications and data must be securely shared between subsets thereof.

5.12 Conclusions

We formulated the notion of *Privacy-from-Birth* (*PfB*) and proposed *VERSA*: a formally verified architecture realizing *PfB*. *VERSA* ensures that only duly authorized software can

access sensed data even if the entire software state is compromised. To attain this, *VERSA* enhances the underlying MCU with a small hardware monitor, which is sufficient to achieve *PfB*. The experimental evaluation of *VERSA* publicly available prototype [12] demonstrates its affordability on a typical low-end IoT MCU: TI MSP430.

Chapter 6

CASU: Compromise Avoidance via Secure Update for Low-end Embedded Systems

Abstract

Proliferation of runtime attacks that introduce malicious code (e.g., by injection) into embedded devices – referred to as *code injection attacks* – has prompted a range of mitigation techniques. While $\mathcal{RA}$ schemes detect such attacks, they require $\mathcal{V}$rf to explicitly initiate $\mathcal{RA}$ request, based on some unclear criteria. Thus, in case of prover's compromise, $\mathcal{V}$rf only learns about it upon the next $\mathcal{RA}$ instance. While sufficient for compromise detection, some applications would benefit from a more proactive, prevention-based approach.

To this end, we construct *CASU*: Compromise Avoidance via Secure Updates. *CASU* is an inexpensive hardware/software co-design enforcing: (i) runtime software immutability, thus precluding any illegal software modification, and (ii) authenticated updates as the sole means of modifying software. In CASU, a successful $\mathcal{RA}$ instance serves as a proof of successful update, and continuous subsequent software integrity is implicit, due to the runtime immutability guarantee. This obviates the need for $\mathcal{RA}$ in between software updates and leads to unobtrusive integrity assurance with guarantees akin to those of prior $\mathcal{RA}$ techniques, with better overall performance.

Research presented in this chapter appeared in the Proceedings of the 41st International Conference on Computer-Aided Design (ICCAD 2022) [50].

6.1 Introduction

Code injection attacks [65, 125, 45, 105] represent a real and prominent threat to low-end devices. Embedded systems software is mostly written in C, C++, or Assembly – languages that are very prone to errors. Code injection attacks exploit these errors to cause buffer overflows and inject malicious code into the existing software or somewhere else in the device memory.

Previous results considered such attacks in low-end devices and proposed $\mathcal{R}$A [62, 103, 47, 21, 30, 83], as well as proofs of remote software updates and memory erasure [48, 20, 27]. However, they have considerable runtime costs since they require computing a cryptographic function (usually, a Message Authentication Code (MAC)) over the entire software. A recent result, *RATA* [51], minimized the cost of $\mathcal{R}$A by measuring a constant-size memory region that reflects the time of the last software modification (legal or otherwise). *RATA* achieved that by introducing a hardware security monitor that securely logs each modification time to that region.

Regardless of their specifics, $\mathcal{R}$A techniques only detect code modifications **after the fact**. They cannot prevent them from taking place. Hence, there could be a sizeable window of time between the initial compromise and the next $\mathcal{R}$A instance when the compromise would be detected.

To this end, the goal of this chapter is to take a more proactive, prevention-based approach to avoid potential compromise. It constructs *CASU*: Compromise Avoidance via Secure Update, which consists of two main components. First is a simple hardware security monitor that is formally verified. It performs two functions: (1) blocks all modifications to the specific program memory region where the software resides, and (2) prevents anything stored outside that region from executing. It runs independently from (in parallel with) the MCU core, without modifying the latter. This thwarts all code injection attacks. However, it is

unrealistic to prohibit all modifications to program memory, since genuine software updates need to be installed during the device's lifetime. Otherwise, the software could be housed in ROM or the entire device would function as an ASIC (Application Specific Integrated Circuit). Therefore, *CASU* second component is a secure remote software update scheme.

The key benefit of *CASU* is maintaining constant software integrity without repeated $\mathcal{R}$A measurements while allowing genuine secure software updates. Specifically, it guarantees that, between any two successive secure updates, device software is immutable. However, the device's liveness can be ascertained at any time by repeating the latest update, which essentially represents $\mathcal{R}$A.

6.1.1 Contributions

The intended contributions of *CASU* are:

1. A tiny formally verified hardware monitor that guarantees benign (authorized) software immutability and prevents the execution of any unauthorized code.
2. A scheme to enable secure software updates when authorized by a trusted 3rd party.
3. An open-source *CASU* prototype built atop a commodity low-end MCU to demonstrate its low cost and practicality.

6.2 Background

This section gives a brief overview of TOCTOU attacks and *RATA*, which mitigates such attacks.

6.2.1 TOCTOU Attacks & *RATA*

Traditional $\mathcal{RA}$ techniques have shared a common limitation: they measure the state of $\mathcal{Prv}$ executables at the time when $\mathcal{RA}$ is executed by $\mathcal{Prv}$. They provide no information about $\mathcal{Prv}$ executables **before** $\mathcal{RA}$ measurement or its state in **between** two consecutive $\mathcal{RA}$ measurements. This problem is referred to as *Time-Of-Check Time-Of-Use* or $\mathcal{RA}$-TOCTOU. In practice, $\mathcal{RA}$-TOCTOU leaves devices vulnerable to transient malware that erases itself after completing its tasks, leaving no detectable traces.

RATA [51] solves $\mathcal{RA}$-TOCTOU with a minimal (formally verified) hardware component to additionally provide historical context about the state of $\mathcal{Prv}$ program memory. This is achieved via securely logging the time when the latest program memory modification was performed. This time is securely logged to a protected memory region called Latest Modification Time (LMT) buffer that can not be modified by any software. LMT is also covered by the $\mathcal{RA}$ function. Therefore, in *RATA* an $\mathcal{RA}$ result tells $\mathcal{Vrf}$ about the current state of $\mathcal{Prv}$ and "since when" this has been the state of $\mathcal{Prv}$. This feature was integrated seamlessly into *VRASED* and the composition was shown to be secure against TOCTOU attacks (see [51] for a detailed definition).

RATA is implemented as a hardware module behaving as follows:

1. It monitors a set of MCU signals and detects whenever any location within an attested memory region AR is written.
2. Whenever a modification in AR is detected, *RATA* logs the timestamp by reading the current time from the real-time clock (RTC) and storing it in a fixed memory location, called Latest Modification Time (LMT).
3. In the memory layout, LMT is stored in the same region as AR and enforced by *RATA* to be read-only for all software executing on the MCU, and for DMA.

In solving the $\mathcal{RA}$-TOCTOU problem, *RATA* also obviates the need to compute $\mathcal{RA}$ the entirety of AR, significantly reducing $\mathcal{RA}$ execution time on $\mathcal{P}\mathsf{rv}$. When $\mathcal{V}\mathsf{rf}$ already knows AR contents from a previous $\mathcal{RA}$ result, it suffices to verify that AR has not changed. Such a proof can be obtained by attesting only LMT, instead of AR in its entirety. Therefore $\mathcal{RA}$ time complexity is reduced from linear on the size of AR to a constant on the size of LMT, i.e., 32 bytes.

In this chapter, unlike *RATA*, *CASU* actively **prevents** any modification to PMEM at runtime, unless it is a securely and causally authorized (by the trusted $\mathcal{V}\mathsf{rf}$) software update.

6.3 *CASU* Scheme & Assumptions

6.3.1 Basics

Similar to the typical $\mathcal{RA}$ setting, *CASU* involves a low-end MCU ($\mathcal{P}\mathsf{rv}$) and verifier ($\mathcal{V}\mathsf{rf}$). The latter is a trusted higher-end device, e.g., a laptop, a smartphone, a smart home gateway, or a device manufacturer's back-end server. $\mathcal{V}\mathsf{rf}$ is responsible for initiating each software update request, verifying whether the update was successful, and keeping track of the latest successfully confirmed software update. We assume a single $\mathcal{V}\mathsf{rf}$ for a given $\mathcal{P}\mathsf{rv}$. Also, $\mathcal{P}\mathsf{rv}$ and $\mathcal{V}\mathsf{rf}$ are assumed to share a master secret key ($\mathcal{K}$) installed on $\mathcal{P}\mathsf{rv}$ at manufacturing time. Our discussion focuses on the symmetric key setting, which is more practical for low-end MCUs. Nonetheless, the use of public-key cryptography is possible with some cosmetic changes to *CASU*, provided that $\mathcal{P}\mathsf{rv}$ has sufficient computing capabilities.

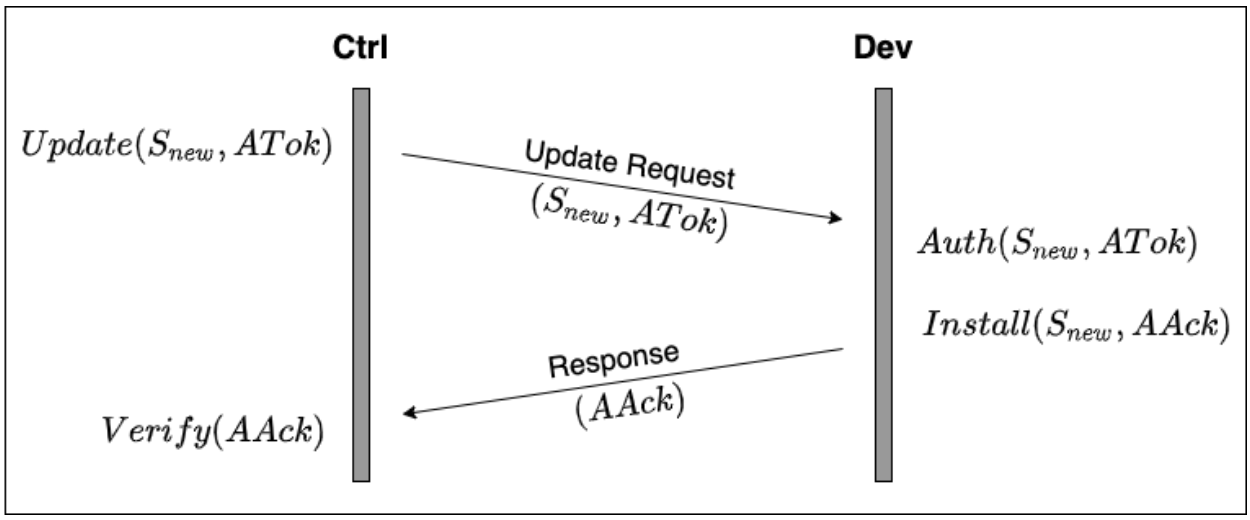

Figure 6.1: *CASU* Secure Update Protocol.

6.3.2 Secure Update Overview

At the time of its initial deployment, $\mathcal{V}$rf is assumed to know the software state ($\mathcal{S}_{\mathsf{old}}$) of $\mathcal{P}$rv. When $\mathcal{V}$rf later wishes to update this software, it issues an update request, denoted by $\mathbf{Update}^{\mathcal{V}\mathsf{rf}}$, to $\mathcal{P}$rv. This request carries the new software $\mathcal{S}_{\mathsf{new}}$ and a fresh authentication token ATok, based on $\mathcal{S}_{\mathsf{new}}$.

When $\mathcal{P}$rv receives an $\mathbf{Update}^{\mathcal{V}\mathsf{rf}}$, $\mathcal{S}_{\mathsf{old}}$ invokes *CASU*, which handles the update process in two steps: (1) $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$ verifies that ATok is a fresh and timely token that corresponds to $\mathcal{S}_{\mathsf{new}}$, and (2) if the first step succeeds, $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ replaces $\mathcal{S}_{\mathsf{old}}$ with $\mathcal{S}_{\mathsf{new}}$ and generates an authenticated acknowledgment (AAck). At this point, *CASU* terminates and control is given to $\mathcal{S}_{\mathsf{new}}$ which must send AAck to $\mathcal{V}$rf.

Upon receiving AAck, $\mathcal{V}$rf executes the Verify procedure to check whether the AAck is a valid confirmation for the outstanding $\mathbf{Update}^{\mathcal{V}\mathsf{rf}}$. If no AAck is received, or if AAck verification fails, $\mathcal{V}$rf assumes a failed update. Figure 6.1 illustrates the interaction between $\mathcal{V}$rf and $\mathcal{P}$rv. Protocol details are described in Section 6.4 below.

6.3.3 Adversary Model

We consider an adversary, $\mathcal{A}\mathsf{dv}$, that controls the entire memory state of $\mathcal{P}\mathsf{rv}$, including PMEM (flash) and DMEM (RAM). It can attempt to write, read or execute any memory location. It can also attempt to remotely launch code injection attacks to modify $\mathcal{P}\mathsf{rv}$ software. It may also divert the execution control-flow to ignore update requests, as well as attempt to extract any $\mathcal{P}\mathsf{rv}$ secrets or forge update confirmations.

Furthermore, $\mathcal{A}\mathsf{dv}$ can configure DMA controllers on $\mathcal{P}\mathsf{rv}$ to read/write to any part of the memory while bypassing the CPU. It can induce interrupts in an attempt to pause the update procedure, modify any part of the old or new software versions, or cause inconsistencies or race conditions. It might also eavesdrop on or interfere with network traffic between $\mathcal{V}\mathsf{rf}$ and $\mathcal{P}\mathsf{rv}$, in a typical Dolev-Yao manner [59].

As common in most related work, physical attacks requiring adversarial presence are considered out of scope. This includes both non-invasive and invasive physical attacks. The former describes attacks whereby $\mathcal{A}\mathsf{dv}$ physically reprograms $\mathcal{P}\mathsf{rv}$ software using direct/wired interfaces, such as USB/UART, SPI, or I2C. The latter refers to inducing hardware faults, modifying code in ROM, extracting secrets via physical side-channels, and tampering with hardware. Protection against non-invasive attacks can be obtained via well-known features, such as a secure boot. (see Section 6.4.3). Whereas, protection against invasive attacks can be obtained via standard tamper-resistant techniques [110].

6.4 *CASU* Design

One of *CASU* main features is the prevention of all unauthorized software modifications to $\mathcal{P}\mathsf{rv}$ software. As mentioned earlier, the former can be trivially achieved by making all $\mathcal{P}\mathsf{rv}$ software read-only, or by making $\mathcal{P}\mathsf{rv}$ an ASIC. However, this precludes all benign

(authorized) updates. Therefore, it is essential to have a secure update mechanism. The term "authorized" refers to software installed on $\mathcal{P}$rv physically at manufacture or deployment time, as well as each subsequent version installed via update request by $\mathcal{V}$rf.

From $\mathcal{V}$rf perspective, *CASU* guarantees that, once installed, authorized software on $\mathcal{P}$rv remains unchanged until the next $\mathcal{V}$rf-initiated successful secure update. This is achieved via three features:

1. *Authorized Software Immutability:* Except via a secure update (implemented within *CASU* trusted code), authorized software cannot be modified.
2. *Unauthorized Software Execution Prevention:* Only the memory containing the (immutable) authorized software is executable.
3. *Secure Update:* $\mathcal{V}$rf is the only entity that can authenticate $\mathcal{P}$rv to install new software. After an update, the previous version of the installed software is no longer authorized.

The first two features are realized by a hardware module, *CASU-HW*, that runs in parallel with the CPU. It monitors a few CPU hardware signals and triggers an MCU reset if any violation is detected. The third feature is realized by a trusted code base (TCB), *CASU-SW*, that extends *VRASED* to authenticate incoming update requests containing new software to be installed ($\mathcal{S}_{\mathsf{new}}$) and an authorization token (ATok) that must be issued by $\mathcal{V}$rf using the key $\mathcal{K}$ pre-shared with *CASU* module in $\mathcal{P}$rv. If ATok matches $\mathcal{S}_{\mathsf{new}}$, then *CASU-SW* installs $\mathcal{S}_{\mathsf{new}}$ on $\mathcal{P}$rv and produces an authenticated AAck, attesting to $\mathcal{V}$rf that a successful update occurred on $\mathcal{P}$rv.

Figure 6.2 depicts *CASU* software execution flow. After each boot or reset, it executes authorized software that was previously installed (either physically embedded or via *CASU* Secure Update). In this state, *CASU-HW* ensures software immutability and execution prevention of anything else. However, when an update request is received, *CASU-SW* must be invoked to securely apply the update and re-configure *CASU-HW* to protect the memory

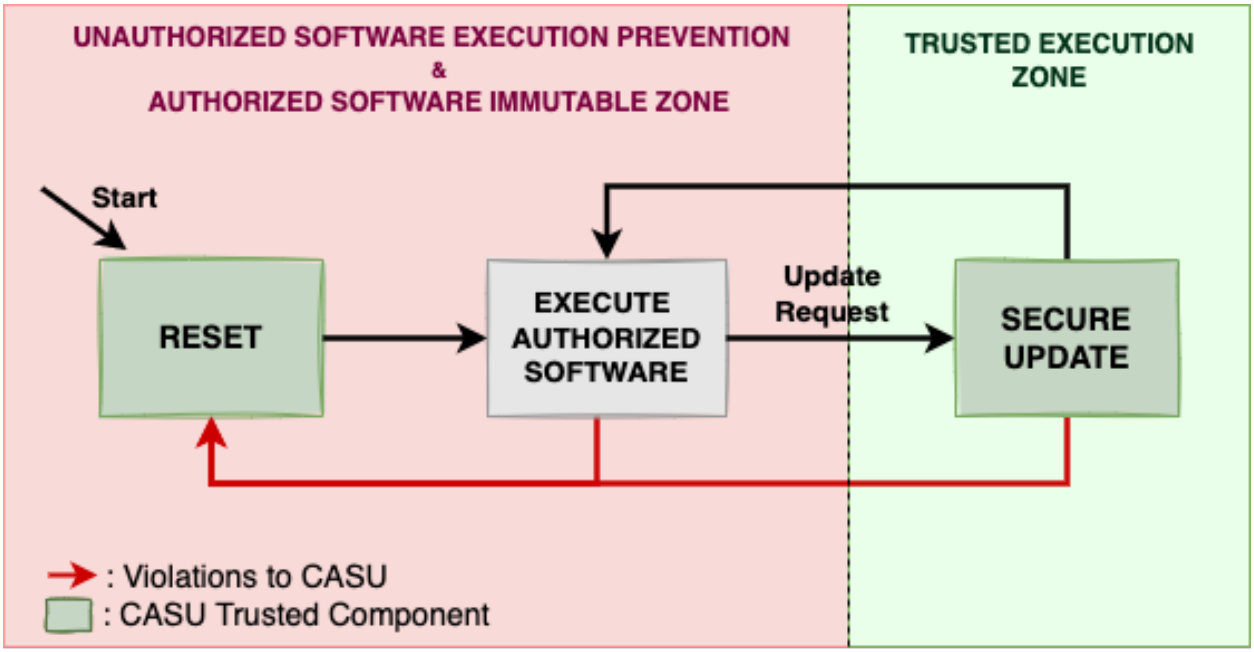

Figure 6.2: *CASU* Software Execution Flow.

region where $\mathcal{S}_{\text{new}}$ is installed. Note that the update cannot be performed without invoking *CASU-SW* due to the immutability guarantee.

Table 6.1 summarizes MCU hardware signals and memory regions relevant to *CASU*. Figure 6.3 illustrates the *CASU* architecture: (1) *CASU-HW* prevents modification of memory regions in gray and prevents execution of all other memory, while (2) *CASU-SW* resides in the ROM; it contains a bootloader and subroutines related to *CASU* secure update. We describe these features in detail in the rest of this section.

6.4.1 *CASU-HW*: Hardware Security Monitor

CASU-HW monitors PC, W_{en}, D_{addr}, DMA_{en}, DMA_{addr} to detect illegal writes or execution. When a *violation* is detected, *CASU-HW* activates the **Reset** signal. To simplify notation when describing *CASU-HW* properties, we define the following macro:

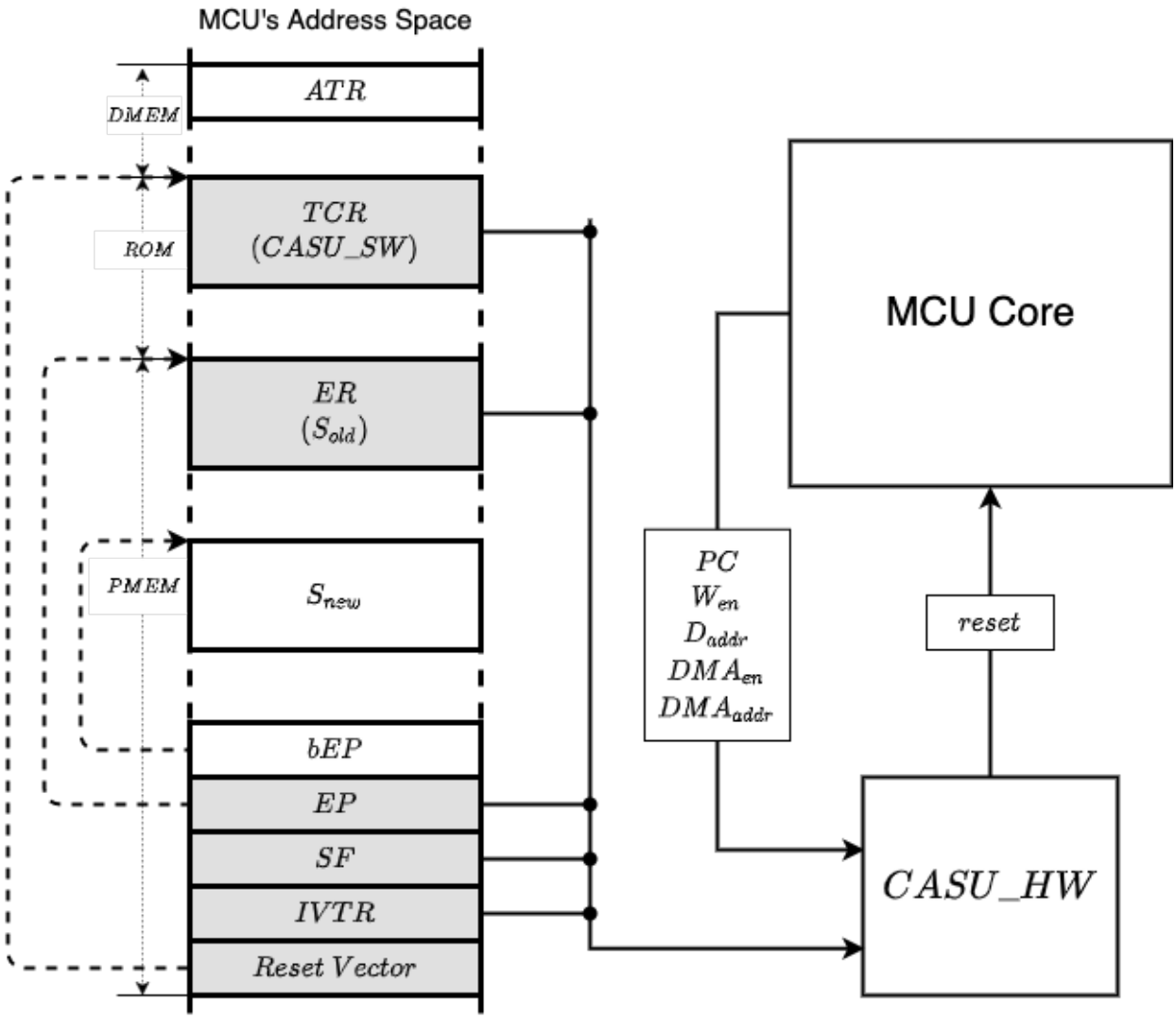

Figure 6.3: *CASU* System Architecture.

Table 6.1: Notation Summary

Notation	Description
PC	Program Counter, points to the current instruction being executed
W_{en}	1-bit signal that indicates if MCU core is writing to memory
D_{addr}	Memory address where the MCU core is currently accessing
DMA_{en}	1-bit signal that indicates if DMA is active
DMA_{addr}	Memory address being accessed by DMA, when active
Reset	Signal that reboots the MCU when set to logic '1'
TCR	Trusted Code Region, a fixed ROM region storing *CASU-SW*
ER	Executable Region, a configurable memory region where authorized software is stored; $ER = [ER_{min}, ER_{max}]$, where ER_{min} and ER_{max} are the boundaries of ER
EP	Executable Pointer, a fixed memory region storing current values of ER_{min} and ER_{max}
bEP	Buffer Executable Pointer, a fixed memory location used to save the boundaries of the memory region storing new software $\mathcal{S}_{\mathsf{new}}$.
ATR	Fixed memory buffer from which $\mathbf{Auth}^{\mathcal{P}_{\mathsf{rv}}}$ reads ATok and also where $\mathbf{Install}^{\mathcal{P}_{\mathsf{rv}}}$ outputs AAck
$IVTR$	Reserved memory region for the MCU's IVT
SF	Fixed memory region where *Status* flag is stored; *Status* is used by *CASU-SW* for consistency.

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

i represents a memory address. $Mod\_Mem\ (i)$ is true whenever the MCU core or the DMA is writing to i. When representing a write within some contiguous memory region (with multiple addresses) $M = [M_{min}, M_{max}]$, we "abuse" the notation as $Mod\_Mem\ (M)$. To denote that a write has occurred within one of the multiple contiguous memory regions, e.g., when a write happens to some address within M_1 or M_2, we say $Mod\_Mem\ (M_1, M_2)$.

Authorized Software Immutability

Software authorized by *CASU*, including any ISRs, is located in the contiguous memory segment ER. The pointer EP stores the boundaries that define ER, i.e., ER_{min} and ER_{max}. *CASU-HW* monitors EP to locate the currently authorized software and enforce its rules based on this region. Write attempts to EP are also monitored and only allowed when performed by *CASU-SW*, preventing malicious changes to EP that could misconfigure the range of ER, leading *CASU-HW* to enforce protections based on the incorrect region. ER is configurable to give *CASU-SW* flexibility to change the location and size of authorized software, instead of fixing $\mathcal{S}_{\mathsf{new}}$ to the same location and size of $\mathcal{S}_{\mathsf{old}}$, as software versions vary in size. *CASU-HW* also protects memory regions SF and $IVTR$. SF is used during a secure update, described in Section 6.4.2. Since ISRs are a part of ER, IVT must be protected to maintain the integrity of interrupt handling during authorized software execution.

Incidentally, Authorized Software Immutability also prohibits self-modifying code, i.e., code in ER writing to ER, to prevent code injection attacks within ER.

Unauthorized Software Execution Prevention

Authorized Software Immutability:

$$[Mod\_Mem(ER, EP, SF, IVTR) \wedge (PC \notin TCR)] \rightarrow \textbf{Reset} \quad (6.1)$$

Unauthorized Software Execution Prevention:

$$[(PC \notin ER) \wedge (PC \notin TCR)] \rightarrow \textbf{Reset} \quad (6.2)$$

Figure 6.4: *CASU-HW* Security Properties.

Only authorized software (located in ER) or *CASU-SW* (located in TCR) are allowed to be executed on $\mathcal{P}\mathsf{rv}$. Since ER is configurable via EP after a secure update, *CASU-SW* re-configures EP to allow execution of $\mathcal{S}_{\mathsf{new}}$ from the new ER location.

6.4.1.1 *CASU-HW* Properties Formally

Figure 6.4 formalizes the aforementioned *CASU-HW* security properties using proportional logic. Note that these properties must hold at all times. Equation 6.1 states that any modification to ER, EP, SF, and $IVTR$– when a program other than *CASU-SW* ($PC \notin TCR$) is executing – causes a **Reset**. Equation 6.2 states that MCU cannot execute programs other than those in ER and TCR. If PC points to any other memory location, the MCU is reset.

6.4.2 *CASU* Secure Update

Recall (from Section 6.3.2) that *CASU* Secure Update implements: ($\textbf{Update}^{\mathcal{V}\mathsf{rf}}$, Verify) on $\mathcal{V}\mathsf{rf}$ and ($\textbf{Auth}^{\mathcal{P}\mathsf{rv}}$, $\textbf{Install}^{\mathcal{P}\mathsf{rv}}$) on $\mathcal{P}\mathsf{rv}$. At a high level, there are two ways of implementing it on $\mathcal{P}\mathsf{rv}$.

1. Download $\mathcal{S}_{\mathsf{new}}$ to DMEM (RAM), i.e., the stack or heap of the current software ($\mathcal{S}_{\mathsf{old}}$),

and invoke $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$. If it succeeds, $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ overwrites ER with $\mathcal{S}_{\mathsf{new}}$ and updates EP. This is problematic, because, if a reset occurs in the middle of $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ execution, then ER containing $\mathcal{S}_{\mathsf{old}}$ would be partially overwritten and $\mathcal{S}_{\mathsf{new}}$ in the DMEM would be lost as a consequence of the reset. This would leave $\mathcal{P}\mathsf{rv}$ software in a corrupted state.

2. Download $\mathcal{S}_{\mathsf{new}}$ to PMEM (flash) and invoke $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$. If $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$ succeeds, $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ updates EP to the location where $\mathcal{S}_{\mathsf{new}}$ resides. This is generally safer since $\mathcal{S}_{\mathsf{new}}$ and $\mathcal{S}_{\mathsf{old}}$ reside in two separate flash memory regions. If the installation is interrupted by a reset, *CASU-SW* can re-invoke $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ to complete the installation. However, this requires $\mathcal{P}\mathsf{rv}$ PMEM to be sufficiently large to accommodate both $\mathcal{S}_{\mathsf{new}}$ and $\mathcal{S}_{\mathsf{old}}$, i.e., at least double the size of ER. We believe that this is a realistic assumption. The size of flash memory on our targetted devices is at least $8KB$, whereas the typical binary size is usually under $2KB$.

Construction 6.1 shows the whole scheme. Recall that *CASU-SW* is immutable (being in ROM). Its functionality is described below.

6.4.2.1 $\mathbf{Update}^{\mathcal{V}\mathsf{rf}}$

Secure update requires for any software $\mathcal{S}_{\mathsf{new}}$ to be installed on $\mathcal{P}\mathsf{rv}$ to adhere to the following format $\mathcal{S}_{\mathsf{new}} := (L_{\mathcal{S}_{\mathsf{new}}}||V_{\mathcal{S}_{\mathsf{new}}}||N_{\mathcal{S}_{\mathsf{new}}}||BIN_{\mathcal{S}_{\mathsf{new}}}||IVT_{\mathcal{S}_{\mathsf{new}}})$, where $L_{\mathcal{S}_{\mathsf{new}}}$, $V_{\mathcal{S}_{\mathsf{new}}}$, $N_{\mathcal{S}_{\mathsf{new}}}$ is the $\mathcal{S}_{\mathsf{new}}$ *header* consisting of its size, version number, and a random nonce, respectively. $BIN_{\mathcal{S}_{\mathsf{new}}}$ is the $\mathcal{S}_{\mathsf{new}}$ binary in byte-code that mandatorily includes a **download** and **acknowledge** subroutine that accepts future update requests and replies acknowledgment message back to $\mathcal{V}\mathsf{rf}$. $IVT_{\mathcal{S}_{\mathsf{new}}}$ is the IVT of $\mathcal{S}_{\mathsf{new}}$ that needs to be overwritten to $IVTR$ region so that MCU knows where to jump into the new software when an interrupt is triggered. Another requirement is that $V_{\mathcal{S}_{\mathsf{new}}}$ should always be greater than the version number of the current (or old) software on $\mathcal{P}\mathsf{rv}$. This avoids replay attacks that attempt to trick $\mathcal{P}\mathsf{rv}$ into installing

CONSTRUCTION 6.1. *Let $\mathcal{K}$ is a symmetric key pre-shared between $\mathcal{V}$rf and $\mathcal{P}$rv (protected by VRASED secure architecture). CASU Secure Update scheme defined by* $[\mathbf{Update}^{\mathcal{V}\mathsf{rf}}, \mathbf{Auth}^{\mathcal{P}\mathsf{rv}}, \mathbf{Install}^{\mathcal{P}\mathsf{rv}}, \mathsf{Verify}]$ *is realized as follows:*

1. $\mathbf{Update}^{\mathcal{V}\mathsf{rf}}(\mathcal{S}_{\mathsf{new}}) \rightarrow \mathit{ATok}$*:*
   $\mathcal{V}$rf *generates a tuple* $T := (\mathcal{S}_{\mathsf{new}}, \mathit{ATok})$*, where* $\mathcal{S}_{\mathsf{new}}$ *is the new software and* ATok *is the accompanying authentication token, as follows:*
   (a) *Compiles and generates* $\mathcal{S}_{\mathsf{new}} := (L_{\mathcal{S}_{\mathsf{new}}} || V_{\mathcal{S}_{\mathsf{new}}} || N_{\mathcal{S}_{\mathsf{new}}} || BIN_{\mathcal{S}_{\mathsf{new}}} || IVT_{\mathcal{S}_{\mathsf{new}}})$*, where* $L_{\mathcal{S}_{\mathsf{new}}}$ *is* $\mathcal{S}_{\mathsf{new}}$ *size,* $V_{\mathcal{S}_{\mathsf{new}}}$ *is* $\mathcal{S}_{\mathsf{new}}$ *version number,* $N_{\mathcal{S}_{\mathsf{new}}}$ *is a random nonce,* $BIN_{\mathcal{S}_{\mathsf{new}}}$ *is* $\mathcal{S}_{\mathsf{new}}$ *binary, and* $IVT_{\mathcal{S}_{\mathsf{new}}}$ *is* $\mathcal{S}_{\mathsf{new}}$ *IVT, to be placed in IVTR of* $\mathcal{P}$rv*.*
   (b) *Computes* ATok *using equation 6.3 with the second operand set to:* $0||\mathcal{S}_{\mathsf{new}}$*, where '0' is the direction indicator from* $\mathcal{V}$rf *to* $\mathcal{P}$rv*.*

$$\mathit{ATok} := HMAC(\mathcal{K}, 0||\mathcal{S}_{\mathsf{new}}) \tag{6.3}$$

   $\mathcal{V}$rf *sends* T *to* $\mathcal{P}$rv *for update.*
2. $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}_{\mathsf{new}}, \mathit{ATok}) \rightarrow \perp / \top$*:*
   Upon receiving a tuple $T := (\mathcal{S}_{\mathsf{new}}, \mathit{ATok})$ *from* $\mathcal{V}$rf*,* $\mathcal{S}_{\mathsf{new}}$ *is downloaded at memory region pointed to by* bEP*, and* ATok *is written to* ATR*. Then* $\mathcal{P}$rv *does the following:*
   (a) *If* $V_{\mathcal{S}_{\mathsf{new}}} <= V_{ER}$*, output* $\perp$ *and return to* ER*; otherwise, proceed to the next step.*
   (b) *Computes* σ *using equation 6.4.*

$$\sigma := HMAC(\mathcal{K}, 0||bEP) \tag{6.4}$$

   (c) *If* $\sigma == \mathit{ATok}$*, output* $\top$ *and invoke* $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$*; otherwise, output* $\perp$ *and return to* ER*, where the current software (*$\mathcal{S}_{\mathsf{old}}$*) resides.*
3. $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}(\mathcal{S}_{\mathsf{new}}) \rightarrow \mathsf{AAck}$*:*
   Upon invocation by $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$*, or at boot time, in case Status is equal to* 1*,* $\mathcal{P}$rv *does the following:*
   (a) *Sets Status to* 1 *and updates* EP *with values in* bEP*.*
   (b) *Updates* $IVTR$ *with* $IVT_{\mathcal{S}_{\mathsf{new}}}$*.*
   (c) *Computes* AAck *using equation 6.5 and stores it at* ATR*. In equation 6.5 the second operand is* $1||V_{\mathcal{S}_{\mathsf{new}}}||N_{\mathcal{S}_{\mathsf{new}}}$*, where '1' is the direction indicator from* $\mathcal{P}$rv *to* $\mathcal{V}$rf*.*

$$\mathsf{AAck} := HMAC(\mathcal{K}, 1||V_{\mathcal{S}_{\mathsf{new}}}||N_{\mathcal{S}_{\mathsf{new}}}) \tag{6.5}$$

   (d) *Sets Status to* 0 *and jumps to new* ER*, which is pointed to by the new value in* EP*.*
   $\mathcal{P}$rv *replies to* $\mathcal{V}$rf *with* AAck *indicating successful update.*
4. Verify(AAck) $\rightarrow \perp / \top$*:*
   Upon receiving AAck *from* $\mathcal{P}$rv*,* $\mathcal{V}$rf *does the following:*
   (a) *Computes* γ *using the same equation 6.5.*
   (b) *If* $\gamma ==$ AAck*, outputs* $\top$*; otherwise outputs* $\perp$*.*

Figure 6.5: *CASU* Secure Update.

an old software version that contains vulnerabilities. In case $\mathcal{V}\mathsf{rf}$ wishes to revert to an older version (e.g., due to later-discovered bugs in $\mathcal{S}_{\mathsf{new}}$), it must issue a brand new update request with the older-version software, though with a **new version** number.

$\mathcal{V}\mathsf{rf}$, by invoking $\mathbf{Update}^{\mathcal{V}\mathsf{rf}}$, computes ATok using equation 6.3 and sends $(\mathcal{S}_{\mathsf{new}}, \mathsf{ATok})$ to $\mathcal{P}\mathsf{rv}$.

6.4.2.2 $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$

When $\mathcal{P}\mathsf{rv}$ receives $\mathbf{Update}^{\mathcal{V}\mathsf{rf}}$ with $\mathcal{S}_{\mathsf{new}}$ and ATok, the current **download** subroutine on $\mathcal{S}_{\mathsf{old}}$ in ER accepts and downloads $\mathcal{S}_{\mathsf{new}}$ to an available PMEM slot. It then writes the pointers to $\mathcal{S}_{\mathsf{new}}$ to bEP, buffer Executable Pointer, in PMEM, and writes ATok to ATR. This **download** subroutine should not be a part of *CASU-SW*, as exposing network interfaces directly to trusted parts of the device is hazardous and may result in the exploitation of unknown vulnerabilities in it, leading to key leakage. Hence, even though ER is untrusted, it should be the one receiving the request, because even if it fails to receive or chooses to not call $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$, then AAck is not generated/sent, which is a clear indication to $\mathcal{V}\mathsf{rf}$ that the update was unsuccessful.

To securely verify that $\mathcal{S}_{\mathsf{new}}$ is valid software to be installed on $\mathcal{P}\mathsf{rv}$, $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$ first checks whether the $V_{\mathcal{S}_{\mathsf{new}}}$ is greater than the one of ER, i.e., V_{ER}. If the $V_{\mathcal{S}_{\mathsf{new}}}$ is valid, it invokes *VRASED* as a subroutine to compute σ according to equation 6.4. If σ matches with ATok received from $\mathcal{V}\mathsf{rf}$, then it outputs $\top$ (accept symbol) and further invokes $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ to apply the update. Otherwise, it outputs $\perp$ (reject symbol) and returns to old software at ER without computing any response to be sent back to $\mathcal{V}\mathsf{rf}$.

Note that *CASU-SW* execution is guarded by *CASU-HW* (which inherits *VRASED* hardware properties), i.e., any interrupts or DMA, or any attempts to access the key or any confidential data that *CASU-SW* generates, will be considered as a *violation* and an MCU reset will be

triggered immediately. Also note that if such an abrupt reset occurs, MCU will return to the old software, and eventually $\mathcal{V}\mathsf{rf}$ has to send a new update request. In this new request, $\mathcal{V}\mathsf{rf}$ can use the same version number (but with a different nonce for maintaining freshness) because the previous update was not applied, and thus, the version number of the current software is still old.

6.4.2.3 $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$

Once $\mathcal{S}_{\mathsf{new}}$ is authenticated, $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ is invoked. This is the critical step of *CASU* Secure Update. It is responsible for updating the EP with bEP, $IVTR$ with $IVT_{\mathcal{S}_{\mathsf{new}}}$ and computing authenticated acknowledgment AAck that is to be replied to $\mathcal{V}\mathsf{rf}$. As mentioned in Section 6.4.2.2, if a reset occurs during any of these sub-steps, they have to be repeated from the beginning. This is because, if EP is updated and $IVTR$ is not, vulnerabilities in old ISRs pointed to by the old IVT can be exploited by malware. Furthermore, if EP and $IVTR$ are updated, yet the computation of AAck failed, $\mathcal{V}\mathsf{rf}$ assumes that the update failed and repeats the update request with the same version number (since EP is updated to the new software), and $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$ will fail again. Therefore, all three sub-steps must take place atomically. To this end, *CASU-SW* uses a *Status* flag SF in PMEM, which it sets and unsets, before and after the completion of $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ sub-steps, respectively.

To handle cases when a reset is triggered during $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$, the Reset Vector in $IVTR$ is programmed to start executing from *CASU-SW*. This technique is analogous to having a bootloader. At boot time, *CASU-SW* uses *Status* to determine whether a reset occurred prior to the completion of $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$. If so, *CASU-SW* re-invokes $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ from the beginning.

Finally, $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ computes AAck according to equation 6.5 and writes it to ATR. After generating AAck, *CASU-SW* jumps to new ER. Now, it is the responsibility of the **acknowledge** subroutine in $\mathcal{S}_{\mathsf{new}}$ to reply to $\mathcal{V}\mathsf{rf}$ with AAck.

Acknowledgment Receipt: There are two unlikely cases where $\mathcal{V}$rf may not receive AAck, after being generated by $\textbf{Install}^{\mathcal{P}\text{rv}}$. Firstly, AAck sent by $\mathcal{P}$rv being lost or corrupted in transit. In this case, upon a time-out, $\mathcal{V}$rf re-sends $\textbf{Update}^{\mathcal{V}\text{rf}}$. Since $\textbf{Install}^{\mathcal{P}\text{rv}}$ stores AAck in a dedicated region of DMEM (ATR), **download** in ER checks whether the update request has the same version number as itself and directly replies AAck to $\mathcal{V}$rf, instead of invoking $\textbf{Auth}^{\mathcal{P}\text{rv}}$ again. Secondly, a reset occurring after a successful update and before AAck is sent to $\mathcal{V}$rf. In that case, AAck is lost and, upon a timeout, $\mathcal{V}$rf needs to send a new $\textbf{Update}^{\mathcal{V}\text{rf}}$ with a new version number. The drawback of this approach is that the same update is re-applied, wasting MCU clock cycles. However, the latter case is very rare, and even if it occurs, *CASU-SW* only takes less than a second to re-install $\mathcal{S}_{\text{new}}$ (see Section 6.6.2).

$\mathcal{V}$rf can distinguish between these cases by first re-sending the same $\textbf{Update}^{\mathcal{V}\text{rf}}$. If there is still no response, then AAck is most likely lost due to a reset and $\mathcal{V}$rf must send a new $\textbf{Update}^{\mathcal{V}\text{rf}}$ with a new version number.

There are other ways to mitigate the aforementioned AAck issues. Rather than storing AAck in DMEM, it could be placed into a reserved memory in PMEM to ensure its persistence even if a reset occurs. Now, **download** can always reply with AAck whenever it sees a duplicate request, thus eliminating the cost of re-update. However, this approach requires an additional write to flash, which may be undesirable. Alternatively, we can use a $\mathcal{V}$rf-supplied timestamp instead of a nonce in $\mathcal{S}_{\text{new}}$ and modify $\textbf{Auth}^{\mathcal{P}\text{rv}}$ to accept duplicate requests with a more recent timestamp. This approach does not require any reserved memory (not even in DMEM). However, it incurs runtime overhead every time $\mathcal{V}$rf issues a duplicate request. Each aforementioned alternative has its own benefits and drawbacks. We leave it up to $\mathcal{V}$rf to decide which is most suitable.

Note that none of the above can result in a DoS attack due to multiple requests, because all $\textbf{Update}^{\mathcal{V}\text{rf}}$-s originate from a legit $\mathcal{V}$rf and are verified by $\textbf{Auth}^{\mathcal{P}\text{rv}}$. Moreover, **download**

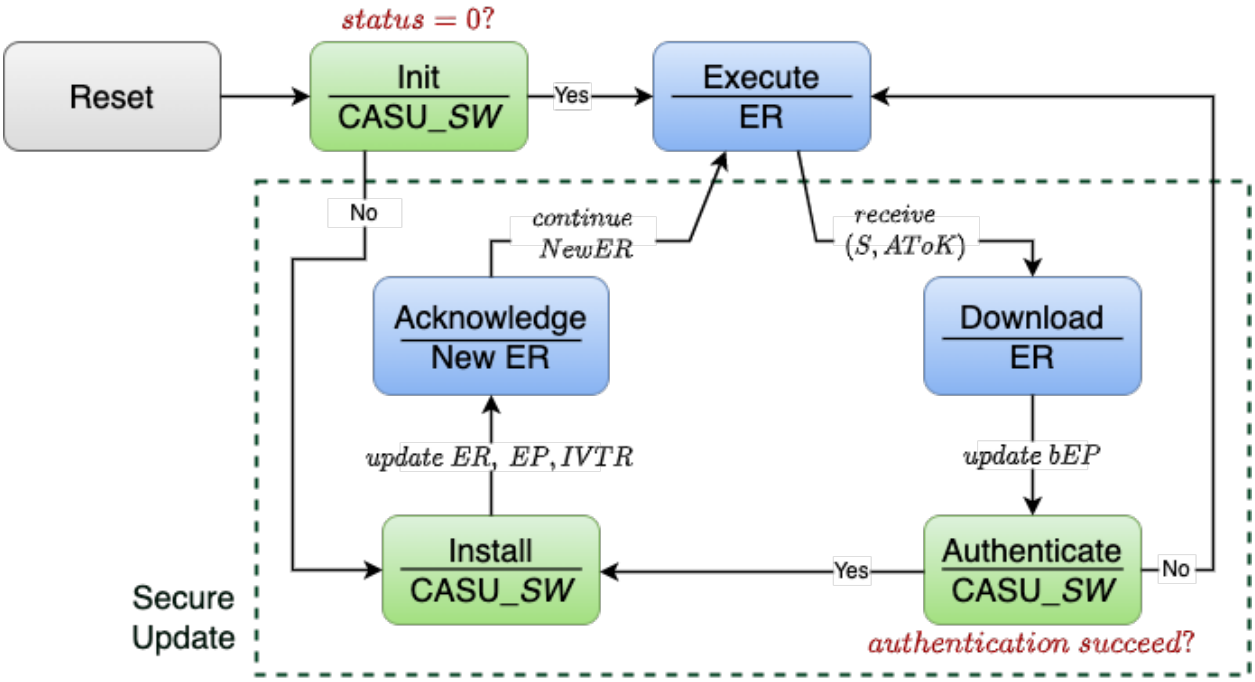

Figure 6.6: Secure Update Workflow: blue and green boxes indicate authorized and trusted execution routines, respectively.

can check the $\mathcal{S}_{\mathsf{new}}$ header to see if the request was already seen, discard the rest of the packets, and simply reply stored AAck to $\mathcal{V}$rf.

6.4.2.4 Verify

Finally, if all goes well, $\mathcal{V}$rf receives an AAck and checks its validity using equation 6.5. If either AAck is invalid or a time-out occurs, $\mathcal{V}$rf assumes that the update failed.

Figure 6.6 depicts the workflow of secure updates. When $\mathcal{P}$rv comes out of reset, it starts executing *CASU-SW*. *CASU-SW* first checks whether *Status* is 1, it invokes $\textbf{Install}^{\mathcal{P}\mathsf{rv}}$ to resume installation of already verified $\mathcal{S}_{\mathsf{new}}$ located at bEP. Otherwise, it jumps to $\mathcal{S}_{\mathsf{old}}$ in ER. Upon receiving $\textbf{Update}^{\mathcal{V}\mathsf{rf}}$, the **download** routine in $\mathcal{S}_{\mathsf{old}}$ accepts and downloads $\mathcal{S}_{\mathsf{new}}$ to an available memory slot in PMEM and stores this address in bEP. $\mathcal{S}_{\mathsf{old}}$ is free to complete its pending tasks before invoking $\textbf{Auth}^{\mathcal{P}\mathsf{rv}}$ in *CASU-SW*. Once, it invokes *CASU-SW*, atomic

execution of $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$ and $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ (if the former succeeds) begins. During $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$, if a *violation* is detected, $\mathcal{P}\mathsf{rv}$ resets and invokes *CASU-SW* with *Status* set to 1, thus invoking $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$ again. After successful completion of $\mathbf{Install}^{\mathcal{P}\mathsf{rv}}$, *CASU-SW* jumps to $\mathcal{S}_{\mathsf{new}}$ in *ER*. Eventually, the **acknowledge** in $\mathcal{S}_{\mathsf{new}}$ replies AAck to $\mathcal{V}\mathsf{rf}$, and continues with its normal execution.

6.4.3 (Optional) *CASU* Secure Boot

CASU's adversary model does not include physical attacks as mentioned in section 6.3.3. *CASU* targets only remote adversaries that try to modify $\mathcal{P}\mathsf{rv}$ software, either by infecting $\mathcal{P}\mathsf{rv}$ with malware that changes the software or by requesting a malicious software update via a forged ATok. We consider such an adversary model because this is the most common type of attack that is viable in pragmatic settings. However, in situations where *CASU* is applied on devices that are prone to physical attacks, specifically of the non-invasive kind, then a secure boot mechanism must be implemented alongside *CASU*.

Secure Boot is a hardware/software component that verifies the integrity of the device's software, before loading and executing it, as soon as the MCU is powered on. After a reset/reboot, the processor is programmed to start executing from Secure Boot. If Secure Boot verifies that the software is in the expected condition, then it continues to invoke this software. Otherwise, it goes into an infinite loop, blocking the device forever until the device is physically reprogrammed with valid software.

To implement Secure Boot as a part of *CASU*, there are two additional changes required. First, *CASU-SW* must be modified to invoke $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$ on *ER*, every time it boots. If $\mathbf{Auth}^{\mathcal{P}\mathsf{rv}}$ verifies that *ER* is unmodified, then *CASU-SW* executes *ER*; otherwise, it blocks $\mathcal{P}\mathsf{rv}$. Second, the hardware of the MCU core must be modified to start from *CASU-SW* deterministically (after Secure Boot). This is because, in the original *CASU* design, boot-

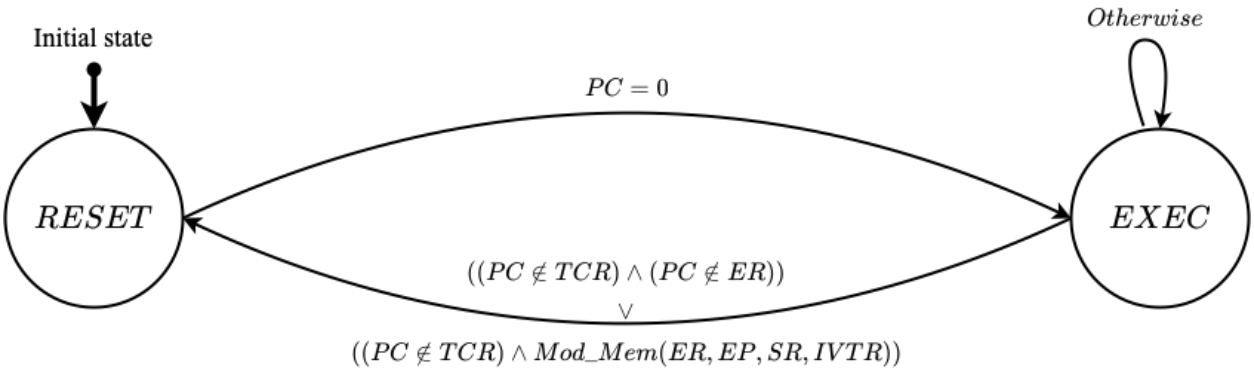

Figure 6.7: FSM of *CASU-HW* Verified Hardware Module.

loading into *CASU-SW* is ensured by IVT guarded by *CASU-HW*. However, if an adversary modifies this IVT by physically reprogramming $\mathcal{P}\mathsf{rv}$, then there is no way *CASU-HW* can prevent this. Therefore, the second change is required to protect against such attacks.

6.5 Implementation

6.5.1 *CASU-HW* Verified Hardware Module

Figure 6.7 presents a hardware FSM formally verified to enforce both properties of Figure 6.4. It is a Mealy FSM, where the output is determined by both the current state and the current input. This FSM takes as input the signals shown in Figure 6.3 and produces a single one-bit output **Reset**. If **Reset** is 1, the MCU core immediately resets.

There are two states in the FSM: *RESET* and *EXEC*. In *RESET*, **Reset** is 1 and remains so until the FSM leaves that state; in other cases, **Reset** is 0. After a reset, as soon as PC reaches 0 (execution is ready to start), the FSM transitions to *EXEC*. While in *EXEC*, the FSM constantly checks for: *(1)* modifications to ER, EP, SF, or $IVTR$, and *(2)* execution attempts outside ER and TCR. In either case, the FSM transitions to *RESET*.

We implement the FSM using Verilog HDL and automatically translate it into Symbolic

Model Verifier (SMV) language using Verilog2SMV [76] tool. Finally, we use the NuSMV Model Checker [40] to generate machine proofs showing that the FSM adheres to the properties in Figure 6.4.

6.5.2 *CASU-SW* Secure Update Routine

CASU-SW implements subroutines `casu_entry`, `casu_authenticate`, `casu_install`, and `casu_exit`.

`casu_entry` is the only legal entry point to *CASU-SW*; it is invoked at boot and during an update. Boot invocation is obtained by setting the IVT reset vector to `casu_entry`, which takes a boolean argument to test whether it was invoked at boot or by ER for an update. In the former case, it checks *Status* to determine whether to invoke `casu_install` in order to resume the unfinished update from the last reset. Otherwise, it calls `casu_exit`, which clears the MCU registers and jumps to the binary in ER. In the latter case, it invokes `casu_authenticate` that checks for the validity of the version number of $\mathcal{S}_{\mathsf{new}}$ at bEP and invokes *VRASED* software to compute HMAC. If the measurement matches `ATok`, `casu_install` is invoked; otherwise, it jumps to `casu_exit`. Finally, `casu_install` updates EP, copies the new IVT to $IVTR$, and computes and stores `AAck` at ATR. It also sets/unsets *Status* to indicate the status of installation to `casu_entry` subroutine, in case of a reset.

CASU-SW is implemented in C with a tiny TCB of $\approx$ 140 lines of code. It uses *VRASED* software, which is implemented using a formally verified cryptographic library, HACL* [139].

6.6 Evaluation

All *CASU* source code and hardware verification/proofs are publicly available at [11]. *CASU* prototype is built on OpenMSP430 [68] and it uses *VRASED* for computing HMAC during Authorize. We use Xilinx Vivado to synthesize an RTL description of *CASU-HW* and deploy it on the Diligent Basys3 board featuring an Artix7 FPGA.

6.6.1 Hardware Overhead

Table 6.2 presents *CASU* hardware overhead compared to unmodified OpenMSP430 and *VRASED*. Similar to prior work [47, 49, 103, 62], we consider additional Look-Up Tables (LUTs) and registers. Compared to *VRASED*, *CASU* only requires 3% (99) additional LUTs and 0.3% (34) additional registers.

Verification Cost: *CASU* was verified using a Ubuntu 18.04 LTS machine running 3.2GHz with 16GB of RAM. Table 6.2 shows verification time and memory. *CASU* requires 95 additional lines of Verilog code to enforce properties in Figure 6.4. The verification cost includes the verification of *VRASED* properties. The time to verify the composite design is under a second and requires 148MB of RAM.

Table 6.2: Hardware Overhead & Verification cost.

Architecture	Hardware		Verification			
	LUTs	Regs	LoC	#(LTLs)	Time (s)	RAM (MB)
OpenMSP430	1859	692	-	-	-	-
VRASED	1902	724	481	10	0.4	13.6
CASU (+*VRASED*)	1958	726	576	12	0.9	148

Comparison with Related Architectures: In Figures 6.8 and 6.9, we compare *CASU* with other low-end MCU security architectures, including *VRASED* [47], *RATA* [51], *APEX* [49], and *PURE* [48], which provide $\mathcal{R}$A-related services. However, recall that, unlike *CASU*, all

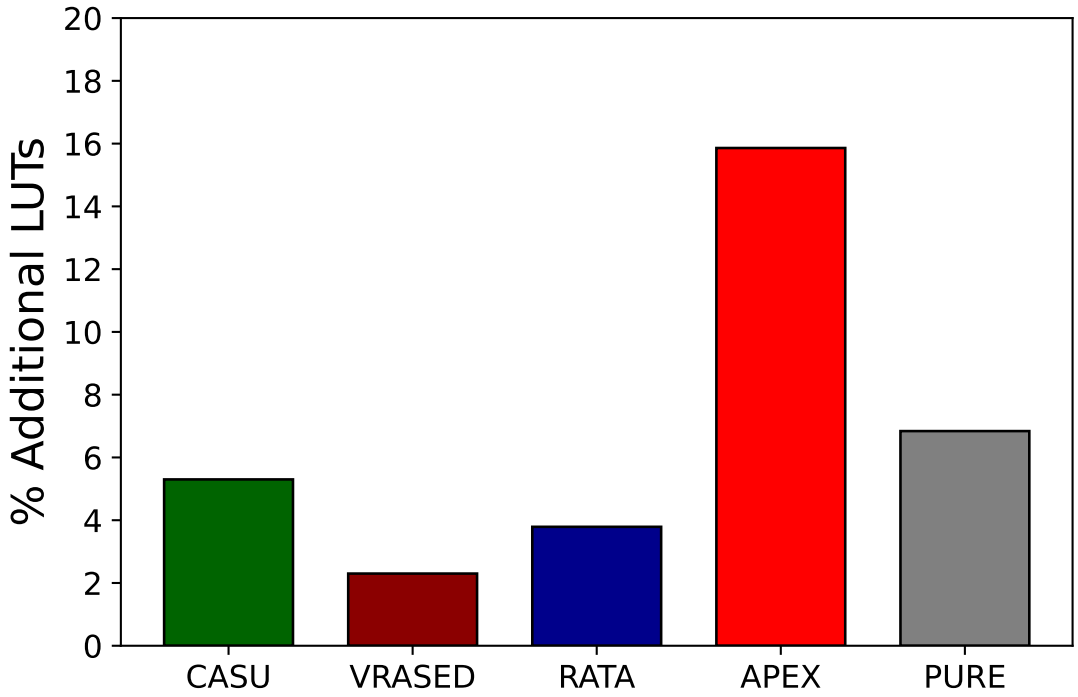

Figure 6.8: *CASU* Additional HW overhead (%) in Number of Look-Up Tables

these other architectures are reactive. As a superset of *VRASED*, *CASU* naturally has a higher overhead. *CASU* and *RATA* have similar overheads, since both monitor memory modifications. Whereas *APEX* and *PURE* enforce additional hardware properties for generating proofs of execution (*APEX*), and proofs of update, reset, and erasure (*PURE*); and thus, they have a higher overhead than *CASU*.

6.6.2 Runtime for Secure Updates

The runtime of *CASU-SW* was evaluated on three sample applications: (1) Blinking LED (250 bytes of binary size) - toggles an LED every half a second, (2) Ultrasonic Ranger (422 bytes) - available at [4] - computes the distance of an obstacle from a moving object, and (3) Temperature Sensor (734 bytes) - available at [3] - measures the temperature of a room. In each case, we measured execution time of `casu_authenticate` and `casu_install`

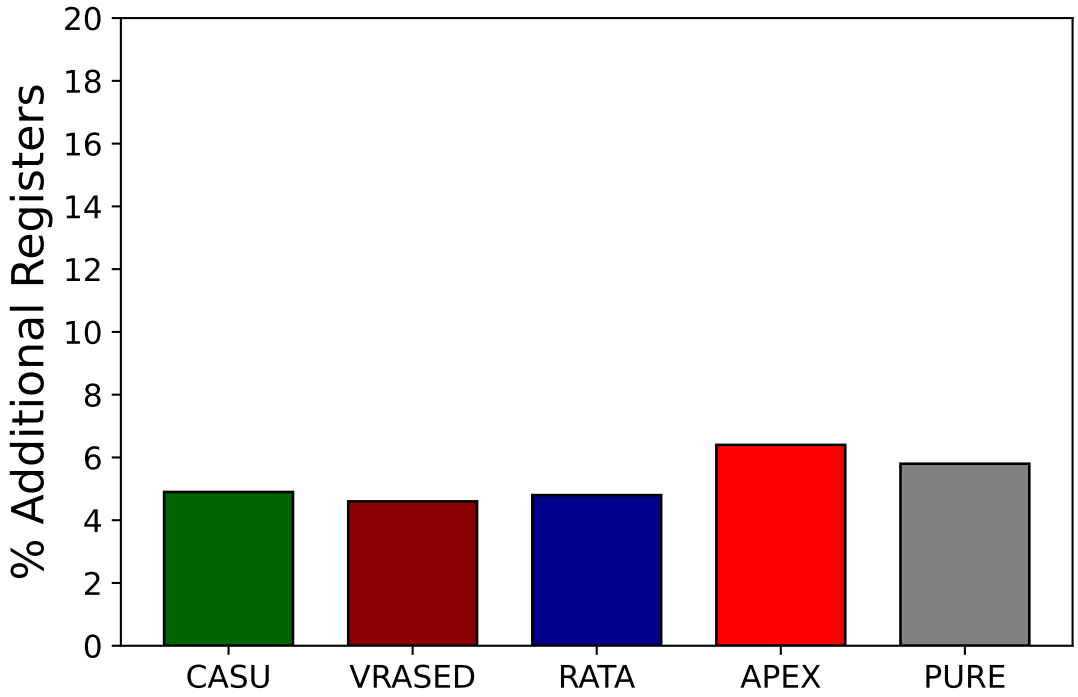

Figure 6.9: *CASU* Additional HW overhead (%) in Number of Registers

– the most time-consuming tasks dominated by HMAC computations. Results are shown in Figure 6.10. `casu_install` runtime is constant because it updates fixed-size memory ranges (including EP, $IVTR$, and SF) and computes HMAC on a fixed-size input. Whereas, `casu_authenticate` scales linearly with $\mathcal{S}_{\text{new}}$ size, over which HMAC is computed. The combined runtime for the worst case (temperature sensor case with 734-byte binary) is $\approx 200ms$, which we consider to be reasonable, considering that updates are infrequent.

Reserved Memory: *CASU* requires 32 bytes of reserved RAM for ATR, 8 bytes of reserved PMEM for EP and bEP, and 1 byte of PMEM for SF. In total, it consumes 41 bytes of additional storage.

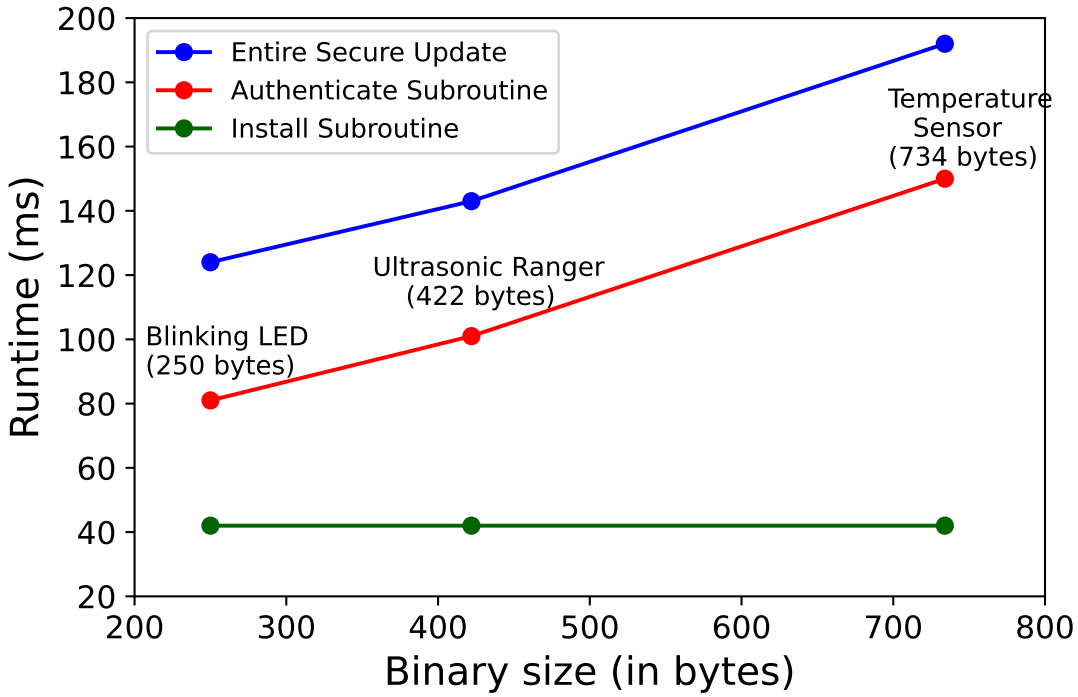

Figure 6.10: Runtime of *CASU-SW* Secure Update

6.7 Conclusions

In this chapter, we designed *CASU*, a prevention-based root-of-trust architecture for low-end MCUs. *CASU* differs from prior work by disallowing illegal software modifications rather than detecting them. *CASU* also prevents execution of any unauthorized software and supports secure software updates. *CASU* is prototyped on OpenMSP430 and its hardware component is formally verified. Experiments show that *CASU* incurs quite low overhead and is thus suitable for resource-constrained low-end IoT devices. Its entire implementation is publicly available at [11].

Chapter 7

Related Work

Abstract

This chapter overviews prior work related to this dissertation. Section 7.1 describes state-of-the-art $\mathcal{R}$A techniques. Next, Section 7.2 discusses prior work on Control-Flow Integrity and $\mathcal{C}$FA techniques. Section 7.3 overviews Data-Flow Integrity and $\mathcal{D}$FA related work. Then, Section 7.4 overviews active RoT designs. Section 7.5 considers some formally verified architectures. Finally, Section 7.6 discusses over-the-air software/firmware updates.

7.1 Prior Work on $\mathcal{R}$A and *PoX*

$\mathcal{R}$A purpose is to measure the software integrity on a given device. There are three main categories of $\mathcal{R}$A techniques: software-based, hardware-based, and hybrid approaches.

7.1.1 Software-based $\mathcal{R}$A

Software-based attestation [81, 119, 117, 114, 66, 88, 69] does not rely on any hardware modifications. However, insecure if $\mathcal{A}$dv can physically re-program $\mathcal{D}$ev (i.e., non-invasive physical $\mathcal{A}$dv). Also, its inability to maintain cryptographic secrets results in strong assumptions about precise timing and constant communication delays between $\mathcal{V}$rf and $\mathcal{P}$rv. These assumptions are generally unrealistic in the IoT ecosystem. Thus, software-based attestation is unsuitable for multi-hop and jitter-prone communication, or settings where a compromised $\mathcal{P}$rv is aided (during attestation) by a more powerful accomplice device. On the other hand, [70] and [21] achieve security against software compromise by relying upon secrets, however, they are not resistant to non-invasive physical attacks. Nonetheless, software-based attestation is the only viable choice for legacy devices that have no security-relevant hardware support.

7.1.2 Hardware-based $\mathcal{R}$A

Hardware-based methods [107, 128, 84, 112, 96, 95, 103] perform $\mathcal{R}$A relying on security provided by either using dedicated hardware components (e.g., TPM [128]) or require substantial changes to the underlying instruction set architecture in order to support execution of trusted software (e.g., Intel SGX [75] or ARM TrustZone [26]). However, the cost of such hardware is normally prohibitive for low-end MCUs. SANCUS [103] is a hardware-based $\mathcal{R}$A architecture for low-end devices (MSP430, in particular). However, its hardware overhead is

more than 100% that of the unmodified MSP430 CPU core.

7.1.3 Hybrid $\mathcal{R}$A

Hybrid attestation techniques [62, 47, 30, 83] leverage minimal hardware support while relying on software to reduce additional hardware complexity. SMART [62] is the first hybrid $\mathcal{R}$A architecture for low-end MCUs, where the MAC (over $\mathcal{P}$rv memory) is implemented in its TCB, while this TCB and the attestation key are protected by minimal hardware extensions. HYDRA [61] relies on a secure boot hardware feature and a formally verified secure microkernel, seL4 [82], while Trustlite [83] modifies MPU and CPU exception hardware to implement $\mathcal{R}$A on the Intel Siskiyou Peak platform. Tytan [30] builds upon Trustlite, extending its capabilities for applications with real-time requirements. Overall, hybrid $\mathcal{R}$A techniques provide a balance between security guarantees and hardware cost, making them suitable for scenarios where low-end devices are deployed remotely.

7.1.4 Temporal Aspects of $\mathcal{R}$A

RATA [51] detects TOCTOU attacks by monitoring software modifications between two $\mathcal{R}$A measurements. Besides TOCTOU attacks, two other temporal aspects are crucial for $\mathcal{R}$A security. The first aspect is temporal consistency [33], which ensures that the $\mathcal{R}$A result reflects an instantaneous and consistent snapshot of attested memory. This consistency is vital because, without it, self-relocating malware could evade detection by relocating itself during $\mathcal{R}$A. Achieving temporal consistency involves enforcing atomic execution of the attestation code or locking attested memory to prevent modifications during $\mathcal{R}$A.

Second, when $\mathcal{R}$A is used on safety-critical and/or real-time devices [32], atomicity requirement might interfere with the real-time nature of $\mathcal{P}$rv application. To address this issue,

SMARM [35] uses probabilistic malware detection. It divides the memory into a set of blocks which are attested in a randomized order. While attesting a block must be atomic, interrupts are allowed between attestation of two blocks. Assuming that malware can not guess the next block, even if interrupts are allowed, malware only has a small probability of avoiding detection. If this procedure is repeated multiple times, such probability becomes negligible. Meanwhile, ERASMUS [34] and SeED [73] are based on $\mathcal{P}$rv self-measurements, where a $\mathcal{P}$rv periodically measures and records its own software state, in order to detect transient malware that infects $\mathcal{P}$rv and leaves before the next $\mathcal{R}$A instance.

7.1.5 *PoX* Related Work

Flicker [96] achieves *PoX* by utilizing TPM-based attestation and sealed storage, combined with late launch support offered by AMD Secure Virtual Machine [19]. This enables isolated code execution and attestation of executed code, as well as its associated inputs and outputs. Sanctum [43], implemented on Rocket RISC-V core, follows a similar approach by instrumenting the enclave code to convey information about its own execution to a remote party. However, both Flicker and Sanctum are designed for high-end devices, *APEX* is the only *PoX* architecture that is tailored for low-end devices that provide *PoX*.

Remark: APEX PoX relies on $\mathcal{R}$A for measuring the attested software and its outputs. In this dissertation, we use *APEX* (for *PoX*) for constructing *Tiny-CFA* and *DIALED* to enable $\mathcal{C}$FA an $\mathcal{D}$FA, and *VRASED* (for $\mathcal{R}$A) for constructing *VERSA* and *CASU* to enable access-control policies for software based on authentication via *VRASED* HMAC computation.

7.2 Mitigation of Control-flow Attacks

Mitigation of runtime exploits is a popular topic and many solutions were proposed [14]. There are two means of addressing control-flow attacks: (a) Control-Flow Integrity aims to prevent control-flow deviations by using prior knowledge of acceptable states of the control-flow traversals, and (b) Control-Flow Attestation detects them by recording and reporting the control-flow path. Below we discuss some prior work in both.

7.2.1 Control-Flow Integrity (CFI)

The main idea of CFI is to derive a program's control-flow graph (CFG) prior to execution, and then monitor its runtime behavior to ensure that the control-flow follows a legitimate path of the CFG; any deviation from the CFG leads to termination of the program. CFI methods [13, 126, 102, 46] primarily aim to find a practical trade-off between runtime overhead and the level of precision. Some work [137, 136] proposed CFI for COTS or legacy binaries. Another relevant scheme is code-pointer integrity (CPI) [86], which focuses on guaranteeing the integrity of code pointers to prevent control-flow hijacking. However, neither CFI nor CPI provides information about the actual control-flow path taken during program execution. Moreover, they do not address non-control data attacks – which corrupt data variables (e.g. loop/if condition variables) that are used to drive the control flow – that execute valid CFG path but still an unexpected path. Furthermore, CFI techniques require extensive hardware support to enable secure shadow stack and constant checking of the destination addresses (to check their validity against valid CFG); hence, unsuitable for low-end MCUs.

7.2.2 Control-Flow Attestation ($\mathcal{C}$FA)

As mentioned in Section 3.2, C-FLAT [15] is the earliest $\mathcal{C}$FA architecture that uses ARM TrustZone Secure World [26]. To remove TrustZone dependence, LO-FAT [55] and Lite-HAX [54] implement CFA using stand-alone hardware modules: a branch monitor and a hash engine. Atrium [135] enhances aforementioned CFA techniques by securing them against physical adversaries that intercept instructions as they are fetched to the CPU. Though less expensive than C-FLAT, such hardware components are still not affordable for low-end MCUs, since their cost (in terms of price, size, and energy consumption) is higher than that of a low-end MCU itself. This is evident from Figures 3.7 and 3.8, which compares hardware costs – in terms of Look-Up Tables (LUTs) and numbers of Registers – of aforementioned $\mathcal{C}$FA techniques and the total hardware cost of the OpenMSP430 core itself.

7.3 Mitigation of Data-only Attacks

There are two mitigation strategies for data-only attacks: (a) Data-Flow Integrity approach that prevents modifications of secure memory/data by unauthorized software snippets, and (b) Data-Flow Attestation technique that instead takes an after-the-fact detection approach.

7.3.1 Data-Flow Integrity (DFI)

Some DFI [57, 98, 56, 99] techniques actively enforce integrity checks to ensure that data flow remains within expected boundaries, thereby preventing unauthorized modifications. Others [17, 37] use static analysis to derive a policy table specifying which memory addresses where each instruction can write. They instrument all memory access instructions to ensure the policy is not violated during runtime. DataShield [36] applies selective protection to

sensitive data based on its data-type; it needs developer annotations to mark such sensitive data. Although effective at preventing data corruption, these techniques tend to incur high runtime overhead due to intercepting and checking all memory accesses.

7.3.2 Data-Flow Attestation ($\mathcal{D}$FA)

Given intensive overhead (in terms of runtime and hardware) of DFI, $\mathcal{D}$FA detects data-only attacks by monitoring the data flow of critical variables during runtime. OAT [124] and LiteHax [54] provide $\mathcal{D}$FA and $\mathcal{C}$FA. While OAT relies on ARM TrustZone-M to record both control-flow path and the critical data variables, LiteHax implements custom hardware on RISC-V Pulpino core [109]. However, similar to aforementioned techniques, they are too costly for low-end MCUs.

It is important to note that $\mathcal{D}$FA techniques, such as OAT, similar to other data-oriented security approaches (e.g., DataShield), require developer annotations to identify critical variables. However, relying on developer annotations introduces the possibility of human error. Hence, there is a need for techniques that minimize reliance on manual annotations and provide robust and automated data-flow attestation mechanisms.

7.4 Other Active RoTs

Passive RoTs detect software compromise by producing an unforgeable proof of $\mathcal{P}$rv state to $\mathcal{V}$rf. In terms of functionality, they implement the following services: (1) $\mathcal{R}$A [62, 103, 47, 21, 30, 83, 128, 81, 119, 117, 114, 66, 96, 112]; (2) *PoX*, $\mathcal{C}$FA, $\mathcal{D}$FA [49, 54, 15, 55, 135, 124, 53, 52, 67]; and (3) proofs of remote software update, erasure, and reset [48, 20, 27]. They *cannot prevent* stated violations.

There are other active RoTs, e.g., Garota [18] is a hybrid architecture that guarantees the execution of critical software even when all the software on the device is compromised. Cider [133] and Lazarus [72] rely on ARM TrustZone or a similar class of MCUs to protect devices from being "bricked", by resetting and updating the device whenever it does not respond to a watchdog timer.

Other hybrid architectures, such as SANCTUM [43] and Notary [28], provide strong memory isolation and peripheral isolation guarantees, respectively. These guarantees are achieved via hardware support or external hardware agents. However, we note that such schemes are designed for high-end computers that support MMUs.

7.5 Formally Verified Systems

Formal Verification provides increased confidence about the correctness of the security techniques' implementations. Several efforts focused on formal verification of security-critical services and systems [139, 87, 82, 29]. seL4[82] is a verified microkernel that provably guarantees memory protection and process isolation. HACL*[139] is a verified cryptographic library proved for functional correctness, memory safety, and secret independence using F* (formal verification) language.

In the space of low-end MCUs, *VRASED* [47] and *RATA* [51] are formally verified hybrid $\mathcal{RA}$ architectures, where the latter one detects TOCTOU attacks. APEX [49] is a verified architecture that provides proofs of remote software execution. PURE [48] offers formally verified proofs of remote update, reset, and erasure. Garota [18] is verified to adhere to the guaranteed execution of critical software when triggered by an interrupt. Another recent result [31] formalized and proved the security of a hardware-assisted mechanism to prevent leakage of secrets through timing side-channels due to MCU interrupts.

7.6 Remote Updates

Remote Over-the-Air (OTA) Updates support the seamless delivery of software updates for IoT devices. Notably, TUF [111] is an update delivery framework resilient to key compromises. Uptane [80] extends TUF for supporting updates for vehicular ECUs. However, both TUF and Uptane require relatively heavy cryptographic operations, unsuitable for *CASU*-targeted low-end devices. ASSURED [27] extends TUF to provide a secure update framework for large-scale IoT deployments. SCUBA [116] uses software-based attestation to identify and patch infected software regions. However, due to the timing assumptions of software-based attestation, it is unsuitable for remote IoT settings. PoSE [106] and AONT [79] use proofs of secure erasure to wipe $\mathcal{P}\mathsf{rv}$ to show that its memory is fully erased and then install new software. However, these schemes are not fault-tolerant and can not retain previous software, in case of reset during erasure or new update installation. Also, an extensive discussion of various software update schemes can be found in [134].

Chapter 8

Conclusions & Future Work

In conclusion, this dissertation addresses security challenges in low-end, resource-constrained MCUs. It identifies shortcomings in current security architectures and presents four RoT architectures: *Tiny-CFA*, *DIALED*, *VERSA*, and *CASU*. *Tiny-CFA* and *DIALED* are passive RoTs that detect runtime software exploits, while *VERSA* and *CASU* are active RoTs that prevent them.

These architectures are constructed using software/hardware co-design, incorporating minimal hardware support to ensure secure isolated execution of their software counterparts. This design principle makes proposed RoTs suitable for low-end MCUs. Another important aspect is the utilization of formally verified hardware support to derive their stated security properties.

The first contribution in Chapter 3, is *Tiny-CFA*, which implements a $\mathcal{C}$FA technique that detects control-flow attacks by measuring the actual control-flow path taken by software. Although there are a few architectures implementing runtime attestation, they rely on expensive hardware support which is not suitable for low-end MCUs. *Tiny-CFA* instead relies on a low-cost *PoX* architecture.

Building upon *Tiny-CFA*, the second contribution is *DIALED*, presented in Chapter 4. It implements $\mathcal{D}$FA to detect both control-flow and data-only attacks, providing comprehensive coverage for all known data-oriented software exploits. By measuring both control- and data-flows, this architecture provides an end-to-end runtime execution report. Also, *DIALED* requires automatic code instrumentation (similar to *Tiny-CFA*), at compile time, whereby all critical data inputs are securely logged.

Next, we presented *VERSA* (discussed in Chapter 5), which thwarts sensor data leakage. Low-end MCUs often collect sensitive personal information from their sensor peripherals. *VERSA* implements a hardware-enforced access control mechanism that prevents unauthorized access to the GPIO-mapped memory of the MCU. The only way to legally access the sensitive memory is to gain authorization from *VERSA* trusted software. *VERSA* hardware is formally verified to protect sensitive memory regions and to provide a secure execution environment for authorized software to process confidential data.

Finally, in Chapter 6, we introduced *CASU*, which prevents runtime code-injection attacks. Similar to *VERSA*, *CASU* implements new verified hardware that prevents modifications to benign (authorized) software and execution of other (unauthorized) software, thereby enforcing runtime integrity. It also offers a secure update framework to remotely update (already authorized) software.

We implement all RoTs on an open-source MSP430 core, a representative MCU for low-end IoT devices, and demonstrate their feasibility using real-world applications. Experimental results show that they have low runtime and hardware overhead.

8.1 Future Work

Limitations & Future Work:

1. Further research on Control-Flow and Data-Only Attacks: The presented RoTs aim to mitigate runtime software exploits. However, none of these approaches can completely prevent control-flow or data-only attacks. Although such attacks can be detected later through control/data-flow attestation techniques, they still succeed in either leaking sensitive information or disrupting safety-critical tasks. In Chapter 7, we discuss techniques that address this limitation on higher-end processors. An interesting future direction is to extend our RoTs to support CFI/DFI by incorporating these ideas or designing new techniques specifically for low-end MCUs. This is particularly challenging because CFI techniques require storing and monitoring all valid control-flow graphs, which can become exponentially large.

2. Support for Interrupt-Aware Architectures: Another limitation is the handling of interrupts in the proposed architectures, except for *CASU*. *Tiny-CFA*, *DIALED*, and *VERSA* handle interrupts by resetting the device. However, for safety-critical applications, these MCUs may benefit from interrupt-friendly architectures. One approach to achieve this is by securely storing the current software context in a protected memory region, executing the corresponding interrupt handler, and then resuming by loading the context back into normal memory. If the MCU's memory is limited, an alternative could be to read/write-protect the current software context in-place. This presents another interesting direction for future work, with the main challenge being the design of verified hardware that requires minimal modifications to implement such functionality.

Other Future Directions:

1. Deploying proposed architectures on commercially available MCUs, such as ARM Cortex-M series [9] MCUs. These MCUs offer slightly higher power and features, including MPUs and (optional) low-end TEEs such as TrustZone-M [90]. Exploring the deployment of the proposed architectures on these MCUs, which are becoming

more affordable and popular, would be interesting.

2. Extending security services to higher-end devices: Another avenue for future research is to explore similar security services for higher-end IoT devices, such as smartwatches and automobile infotainment units. These devices typically host more powerful processors such as ARM Cortex-A series [1] and RISC-V cores [2, 109], which have MMUs and support operating systems with virtualization enabled. Some even have TEEs such as TrustZone-A [89] and Multizone [71]. Leveraging these capabilities and building security architectures that integrate with existing platforms can provide enhanced security for high-end applications.

3. Designing and verifying security architectures using formally verified microkernels: The final future direction involves designing and verifying security architectures for high-end applications using a formally verified microkernel, seL4 [82]. seL4 offers provable memory protection and process isolation guarantees on processors such as the ARM Cortex-A series. Exploring the integration of this secure microkernel with the proposed architectures can provide high-assurance security solutions for mid-to-high-end systems.

Bibliography

[1] Arm cortex-a9. https://developer.arm.com/Processors/Cortex-A9.

[2] SiFive Boards.

[3] Temperature sensor code. https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor.

[4] Ultrasonic ranger code. https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/ultrasonic_ranger.

[5] Avr atmega 1284p 8-bit microcontroller. http://ww1.microchip.com/downloads/en/DeviceDoc/doc8059.pdf, 2009.

[6] *Wireless Sensor and Actuator Networks*, pages 295–300. John Wiley & Sons, Ltd, 2010.

[7] BASHLITE. https://www.enigmasoftware.com/bashlite-malware-hits-one-million-iot-devices/, 2016.

[8] Hajime. https://threatpost.com/mirai-and-hajime-locked-into-\iot-botnet-battle/125112/, 2016.

[9] Arm cortex-m prototyping system. https://www.arm.com/products/tools/development-boards/versatile-express/cortex-m-prototyping-system.php, 2017.

[10] Msp430 flash memory characteristics. https://www.ti.com/lit/an/slaa334b/slaa334b.pdf?ts=1638460551489, 2018.

[11] CASU source code. https://github.com/sprout-uci/CASU, 2022.

[12] VERSA source code. https://github.com/sprout-uci/pfb, 2022.

[13] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In V. Atluri, C. A. Meadows, and A. Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353. ACM, 2005.

[14] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 2009.

[15] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 743–754. ACM, 2016.

[16] A. Acar, H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi, and S. Uluagac. Peek-a-boo: I see your smart home activities, even encrypted! In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 207–218, New York, NY, USA, 2020. Association for Computing Machinery.

[17] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 263–277. IEEE Computer Society, 2008.

[18] E. Aliaj, I. De Oliveira Nunes, and G. Tsudik. GAROTA: generalized active root-of-trust architecture. *CoRR*, abs/2102.07014, 2021.

[19] AMD Technology. AMD64 Architecture Programmer's Manual, Volume 2, 2023.

[20] 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.

[21] 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.

[22] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster. Keeping the smart home private with smart (er) iot traffic shaping. *arXiv preprint arXiv:1812.00955*, 2018.

[23] N. Apthorpe, D. Reisman, and N. Feamster. Closing the blinds: Four strategies for protecting smart home privacy from network observers. *arXiv preprint arXiv:1705.06809*, 2017.

[24] N. J. Apthorpe, D. Reisman, and N. Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. *CoRR*, abs/1705.06805, 2017.

[25] E. Aras, M. Ammar, F. Yang, W. Joosen, and D. Hughes. Microvault: Reliable storage unit for iot devices. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 132–140, 2020.

[26] Arm Ltd. Arm TrustZone. `https://www.arm.com/products/security-on-arm/trustzone`, 2018.

[27] 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.

[28] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 97–113, New York, NY, USA, 2019. Association for Computing Machinery.

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

[30] F. Brasser, B. E. Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 34:1–34:6. ACM, 2015.

[31] 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.

[32] 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.

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

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

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

[36] S. A. Carr and M. Payer. Datashield: Configurable data confidentiality and integrity. In R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 193–204. ACM, 2017.

[37] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In B. N. Bershad and J. C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 147–160. USENIX Association, 2006.

[38] 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.

[39] Y. Cheng, X. Ji, X. Zhou, and W. Xu. Homespy: Inferring user presence via encrypted traffic of home surveillance camera. In *ICPADS*, pages 779–782, 2017.

[40] 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.

[41] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[42] M. Corporation. Mitre att&ck. `https://attack.mitre.org/`, 2015.

[43] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[44] 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.

[45] 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.

[46] J. Criswell, N. Dautenhahn, and V. S. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 292–307. IEEE Computer Society, 2014.

[47] 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.

[48] 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.

[49] 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.

[50] I. De Oliveira Nunes, S. Jakkamsetti, Y. Kim, and G. Tsudik. Casu: Compromise avoidance via secure update for low-end embedded systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[51] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik. On the toctou problem in remote attestation. *CCS*, 2021.

[52] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik. Dialed: Data integrity attestation for low-end embedded devices. 2021.

[53] 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.

[54] 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.

[55] 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.

[56] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In S. J. Eggers and J. R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 103–114. ACM, 2008.

[57] D. Dhurjati and V. S. Adve. Backwards-compatible array bounds checking for C with very low overhead. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 162–171. ACM, 2006.

[58] A. Di Pinto, Y. Dragoni, and A. Carcano. Triton: The first ics cyber attack on safety instrument systems. In *Black Hat USA*, 2018.

[59] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 1983.

[60] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0—a framework for ltl and ω-automata manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, 2016.

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

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

[63] H. Fereidooni et al. Breaking fitness records without moving: Reverse engineering and spoofing fitbit. In *RAID'17*, 2017.

[64] 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.

[65] A. Francillon and C. Castellucia. Code injection attacks on harvard-architecture devices. In *CCS '08*, 2008.

[66] R. W. Gardner, S. Garera, and A. D. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE TIFS*, 2009.

[67] 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.

[68] O. Girard. openMSP430, 2009.

[69] V. D. Gligor and S. L. M. Woo. Establishing software root of trust unconditionally. In *NDSS*, 2019.

[70] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo. PISTIS: Trusted computing architecture for low-end embedded systems. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[71] HexFive. Hexfive multizone security. `https://hex-five.com/`.

[72] M. Huber, S. Hristozov, S. Ott, V. Sarafov, and M. Peinado. The lazarus effect: Healing compromised devices in the internet of small things. In H. Sun, S. Shieh, G. Gu, and G. Ateniese, editors, *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 6–19. ACM, 2020.

[73] 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.

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

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

[76] 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.

[77] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1868–1882. ACM, 2018.

[78] M. Jin, R. Jia, and C. J. Spanos. Virtual occupancy sensing: Using smart meters to indicate your presence. *IEEE Transactions on Mobile Computing*, 16(11):3264–3277, 2017.

[79] G. O. Karame and W. Li. Secure erasure and code update in legacy sensors. In *Trust and Trustworthy Computing*, pages 283–299. Springer International Publishing, 2015.

[80] T. Karthik, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos. Uptane: Securing software updates for automobiles. In *International Conference on Embedded Security in Car*, pages 1–11, 2016.

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

[82] G. Klein, K. Elphinstone, G. Heiser, et al. seL4: Formal verification of an OS kernel. In *ACM SIGOPS*, 2009.

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

[84] 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.

[85] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler. *JEDI*: Many-to-many end-to-end encryption and key delegation for iot. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1519–1536, 2019.

[86] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In J. Flinn and H. Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 147–163. USENIX Association, 2014.

[87] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[88] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *ACM CCS*, 2011.

[89] A. Limited. ARM TrustZone for cortex-a. `https://www.arm.com/technologies/trustzone-for-cortex-a`.

[90] A. Limited. ARM TrustZone for cortex-m. `https://www.arm.com/technologies/trustzone-for-cortex-m`.

[91] H. Lin and N. W. Bergmann. Iot privacy and security challenges for smart home environments. *Information*, 7(3):44, 2016.

[92] Y. Lindell and J. Katz. *Introduction to modern cryptography*, chapter 4.3, pages 109–113. Chapman and Hall/CRC, 2014.

[93] D. W. Loveland. *Automated Theorem Proving: a logical basis*. Elsevier, 2016.

[94] W. Magazine. The botnet that broke the internet isn't going away. https://www.wired.com/2016/12/botnet-broke-internet-isnt-going-away/, 2016.

[95] 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.

[96] 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.

[97] K. L. McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[98] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 245–258. ACM, 2009.

[99] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In J. Vitek and D. Lea, editors, *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, pages 31–40. ACM, 2010.

[100] S. Narain, T. D. Vo-Huu, K. Block, and G. Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 397–413. IEEE, 2016.

[101] A. L. M. Neto, A. L. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. Loureiro, D. F. Aranha, H. K. Patil, 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 CD-ROM*, pages 1–15, 2016.

[102] B. Niu and G. Tan. Per-input control-flow integrity. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 914–926. ACM, 2015.

[103] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. C. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, 2017.

[104] I. D. O. Nunes, S. Hwang, S. Jakkamsetti, and G. Tsudik. Privacy-from-birth: Protecting sensed data from malicious sensors with VERSA. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2413–2429. IEEE, 2022.

[105] OWASP. Owasp top ten. https://owasp.org/www-project-top-ten/, 2021.

[106] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, 2010.

[107] 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.

[108] P. Porambage, M. Ylianttila, C. Schmitt, P. Kumar, A. Gurtov, and A. V. Vasilakos. The quest for privacy in the internet of things. *IEEE Cloud Computing*, 3(2):36–45, 2016.

[109] PULP Platform. PULPino.

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

[111] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, page 61–72. Association for Computing Machinery, 2010.

[112] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1):13 – 22, 2008.

[113] D. Schneider. Jeep Hacking 101. http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101, 2015.

[114] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS*. 2008.

[115] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security (WiSe)*, pages 85–94, Los Angeles, CA, USA, 2006. ACM.

[116] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *In Proceedings of the 5th ACM workshop on Wireless security (WiSe '06)*, page 85–94, 2006.

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

[118] 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.

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

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

[121] I. Spectrum. The real story of Stuxnet. `http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet`, 2013.

[122] statista. "number of internet of things (iot) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030". `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`, 2022.

[123] A. S. A. Sukor, A. Zakaria, N. A. Rahim, L. M. Kamarudin, R. Setchi, and H. Nishizaki. A hybrid approach of knowledge-driven and data-driven reasoning for activity recognition in smart homes. *Journal of Intelligent & Fuzzy Systems*, 36(5):4177–4188, 2019.

[124] 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.

[125] 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.

[126] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In K. Fu and J. Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 941–955. USENIX Association, 2014.

[127] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky. Packet-level signatures for smart home devices. In *Network and Distributed Systems Security (NDSS) Symposium*, volume 2020, 2020.

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

[129] A. Ukil, S. Bandyopadhyay, and A. Pal. Iot-privacy: To be private or not to be private. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 123–124. IEEE, 2014.

[130] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq. The TV is smart and full of trackers: Measuring smart TV advertising and tracking. *Proc. Priv. Enhancing Technol.*, 2020(2):129–154, 2020.

[131] R. H. Weber. Internet of things–new security and privacy challenges. *Computer law & security review*, 26(1):23–30, 2010.

[132] L. Xiong, T. Peng, F. Li, S. Zeng, and H. Wu. Privacy-preserving authentication scheme with revocability for multi-wsn in industrial iot. *IEEE Syst. J.*, 2023.

[133] 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 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1415–1430. IEEE, 2019.

[134] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli. Secure firmware updates for constrained iot devices using open standards: A reality check. *IEEE Access*, pages 71907–71920, 2019.

[135] 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.

[136] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 559–573. IEEE Computer Society, 2013.

[137] M. Zhang and R. Sekar. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 91–100. ACM, 2015.

[138] S. Zheng, N. Apthorpe, M. Chetty, and N. Feamster. User perceptions of smart home iot privacy. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–20, 2018.

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