UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Doctoral Programme in
Information Engineering and Computer Science

On the importance of software-hardware co-design of Trusted Computing architectures and services

Michele Grisafi

Advisor
Prof. Bruno Crispo
Università di Trento

January 2025

Acknowledgements

"Who then shall commence? Ever the one whose
lips first give breath to the question."
— *Unknown*

Time is such a marvelous thing. It slips through your fingers like sand, sometimes rushing, sometimes lingering, and sooner or later your hand is empty and almost five years have passed by. Oh, how have I treasured this handful of sand! Now, as I write the final chapter of this thesis, I find myself searching for the perfect way to conclude it. Perhaps, this is the hardest chapter of all, not for lack of words, but because finding the best way to express my feeling at the end of such a journey is difficult. Yet, like a poet chasing the perfect verse, I will try my best.

In the hope that my *quill* will come to my aid, I'll start by thanking those who made this work possible. First, Bruno Crispo, my supervisor. I still remember one of the best pieces of advice I've ever received: *"It doesn't matter where you do your PhD, just be sure to do it with Bruno"*. I must say that, although it might seem biased, I truly believe that there aren't many advisors like you. It's not just because you provided the right counsel and guidance throughout my academic journey, but because you created a perfect environment for research, filled with incredible people and opportunities. It has never been boring, it has never been stressful, it has always been plenty of fun. All of this was possible thanks to you, and for that, I am deeply grateful.

But whose advice was that? Mahmoud, we may not have walked this entire path together, but we surely made it count! I am sure you know all you did for me, all the help and support. Sure, we collaborated, but more than that we shared a fragment of life. You have been a friend and a colleague, always there despite your own challenges. So, thank you again. And though our path may now diverge, I have a feeling we will collaborate and share more memories, sooner or later.

A heartfelt thank you is due to all of my co-authors and everyone who contributed to my work. I can say that I really enjoyed working with you all, and I hope the feeling is mutual! A special thank you also to Sandro Pinto and Ivan De Oliveira Nunes, whose precious time and feedback on my thesis have been truly appreciated. I sincerely hope you enjoyed reading about my journey, and know that this thesis is what it is also thanks to you. Also to Domenico Siracusa, who has agreed to participate in my committee, hopefully enjoying both the thesis and the defence. Thanks also to all the professors, researchers, students and administratives who accompanied me during these years. I am sure you helped me in some way or the other!

And now, the hardest part. I have never been fond of academic celebrations. I didn't celebrate my bachelor's degree, nor did I celebrate my master's. Partially because I've always found the traditions surrounding these events a bit silly, and partially because I saw them as just another step toward the future. "*What's the big deal about a bachelor degree? There is still the master around the corner!*" "*Why celebrate the end of a master's? There is still a PhD waiting!*" Now, standing at the finish line, on what I believe to be the final step of my formal education, I can't think of a better way to

celebrate than by playing a *Bisca* with you guys. You know who you are, and you know that one of the hardest choices in my life will be parting ways with you. We have shared so many fun moments, so much laughter and so much precious time. Some will call us hard workers and praise our achievements, while others will say all we did was play around and have fun. Well, I don't know whom we should believe, but I wouldn't change a thing! I'm sorry that some of us left sooner than expected (and no, I'm not just talking about Luca and Matteo the palms!), but I cherish every moment we shared. I know I will never find such great working companions anywhere else (but if I do, I'll let you know!). So, good luck to all of you—I am certain that no matter where we go, our friendship will always endure.

Grazie anche a tutta la mia famiglia! Sia a chi mi ha aiutato con la tesi (*Cat, le tue correzioni mi porteranno alla lode, ne sono sicuro!!*), sia a voi altri che mi avete semplicemente supportato (e sopportato) in questi anni. Se sono arrivato alla fine di un dottorato lo devo anche a voi. E per tutti voi altri, amici, compagni di vita e di avventure, sappiate che la serenità con la quale ho affrontato questo percorso è stata possibile anche grazie a voi.

So that is it! These have been the best years of my life and I hope that this small testament might inspire others to follow and start their own research journey. A final thank you to *you*, reader! If I know you, I might have forgotten to thank you, in which case forgive me! If you don't know me, I hope you enjoy this thesis, and feel free to reach out if you have any questions.

Michele Grisafi
Trento, March 2025

Abstract

The increasing heterogeneity and connectivity of computer systems, from low-end embedded devices to high-end servers, present a unique security challenge. The ubiquity of these systems and their role in critical infrastructure has made them prime targets for cyberattacks, with potentially disastrous consequences.

To address these threats, a broad spectrum of security techniques has been developed, combining hardware and software components. Among these, Trusted Computing technologies stand out, providing hardware Roots of Trust (RoTs), such as Trusted Execution Environments (TEEs), and robust software layers to implement foundational security primitives and services. However, these advanced architectures require specific hardware features, leaving devices without such capabilities — particularly resource-constrained low-end devices — vulnerable. For these devices, security must rely more heavily on software-based approaches, which can bridge the gap by providing critical protections in the absence of specialized hardware.

This thesis highlights the need for balanced software-hardware security solutions to accommodate diverse hardware platforms. Focusing on Trusted Computing architectures and services, it proposes several software-hardware co-designs to either create security primitives from scratch, or extend existing ones. The research begins with the development of a novel software-based security stack that operates without relying on hardware security features. This stack, implemented on constrained low-end devices, achieves security guarantees comparable to those of high-end counterparts, even in the absence of fundamental components such as a Memory Protection Unit (MPU). At the core of our stack is a software-based Trusted Computing Base (TCB), functioning as a bare-metal TEE with a self-contained RoT. The TCB enables key security primitives, including memory isolation and privilege separation, and supports Trusted Applications (TAs) such as remote attestation and secure code update.

The TCB is further extended with two key security mechanisms: a control-flow integrity system and a secure checkpointing solution for intermittent computing. Then, this thesis shows an example of software-hardware co-design from the opposite perspective, where a security feature like memory tagging implemented in hardware but not yet adequately used by existing software, can be leveraged to introduce a novel and more advanced exploit mitigation technique.

By addressing the security gap between low- and high-end devices, this research demonstrates how flexible software-hardware architectures can enhance the security of modern multi-device ecosystems.

Keywords

System Security, Embedded Systems, Internet of Things (IoT), Software-based Security, Trusted Execution Environments (TEE), Trusted Computing, Hardware-assisted security, Exploit Mitigation

Contents

List of Tables

List of Figures

List of Acronyms

AES Advanced Encryption Standard.

AP Access Policy.

API Application Programming Interface.

ASLR Address Space Layout Randomisation.

BIOS Basic Input/Output System.

BOP Block-Oriented Programming.

BSL Bootloader.

CFA Control-Flow Attestation.

CFG Control-Flow Graph.

CFI Control-Flow Integrity.

CGC checkpoint Generation Calls.

COTS commercial-off-the-shelf.

CPU Central Processing Unit.

CRC Cyclic Redundancy Check.

CVE Common Vulnerabilities and Exposures.

CVSS Common Vulnerability Scoring System.

DDM Direct Data Manipulation.

DDoS Distributed Denial of Service.

DEP Data Execution Prevention.

DFI Data-Flow Integrity.

DMA Direct Memory Access.

DOP Data-Oriented Programming.

DVR Digital Video Recorder.

ELF Executable and Linkable Format.

EOS Embedded Operative System.

FPGA Field Programmable Gate Array.

GDPR General Data Protection Regulation.

HAL Hardware Abstraction Layer.

ICT Indirect Control Transfer.

IMD Implantable Medical Device.

IoT Internet of Things.

IR Intermediate Representation.

ISA Instruction Set Architecture.

ISR Interrupt Service Routine.

IVT Interrupt Vector Table.

MAC Message Authentiation Code.

MCU Micro Controller Unit.

MMIO Memory Mapped Input/Output.

MMU Memory Management Unit.

MPU Memory Protection Unit.

MPX Memory Protection Extension.

MTE Memory Tagging Extension.

NX Never-eXecute.

OS Operative System.

PAC Pointer Authentication Code.

PC Program Counter.

PMP Physical Memory Protection.

POC Proof of Concept.

PTE Page Table Entry.

RAM Random Access Memory.

REE Rich Execution Environment.

RFID Radio-Frequency Identification.

ROP Return-Oriented Programming.

RoT Root of Trust.

SEV Secure Encrypted Virtualisation.

SFI Software Fault Isolation.

SGX Software Guard Extensions.

SoC System on Chip.

SP Stack Pointer.

SVC SuperVisor Call.

TA Trusted Application.

TBI Top Byte Ignore.

TCB Trusted Computing Base.

TCM Trusted Computing Module.

TDX Trust Domain Extensions.

TEE Trusted Execution Environment.

TPM Trusted Platform Module.

UEFI Unified Extensible Firmware Interface.

XSS Cross-Site Scripting.

Chapter 1

Introduction

1.1 The role of cybersecurity

We have grown so accustomed to technology that we often overlook how dependent we are on it. It has been a driving force that has deeply transformed our society, becoming an integral and omnipresent part of our world. It shapes our environment and significantly impacts our human interactions. Nearly every aspect of daily life has been altered, enhanced, or reimagined to a degree that would have been inconceivable two centuries ago. Much of this progress can be attributed to *computer systems*, which have emerged as a cornerstone of modern society.

Computer systems are omnipresent, forming an intricate network that envelops our society and facilitates nearly every aspect of modern life through devices, applications, and data flows. Smartphones have evolved into indispensable digital assistants, offering millions of services, while artificial intelligence has become an everyday companion. Meanwhile, the Internet of Things (IoT) enables seamless communication among billions of devices. Yet, as society becomes increasingly reliant on this technology, our interactions generate vast amounts of data that contribute to a growing digital ecosystem managed by interconnected computer systems.

Despite their clear benefits, the increasing ubiquity of computer systems and our growing reliance on them raise significant concerns. The CrowdStrike [216] outage underscored just how deeply we depend on these systems, when a simple software bug in a Windows kernel module paralysed critical societal functions, from transportation to healthcare, exposing our vulnerabilities. In such a complex, interconnected world, where a minor bug or cyberattack could paralyse key aspects of life, the importance of cybersecurity becomes undeniable. Without robust cybersecurity measures, our reliance

on computer systems leaves us exposed to catastrophic risks.

Since the advent of the first major cyber attacks (e.g. Morris Worm [192]) cybersecurity has been a fundamental discipline of computer systems. Unfortunately, security has historically been overshadowed by priorities like performance optimisation and user-friendliness.[1] Examining the *cost of security*, we can understand why both providers and consumers might approach security cautiously. While its benefits can be subtle to be perceived, e.g. it is often hard to detect unsuccessful attacks, its downsides are plain: performance downgrade, UI degradation and an increase in both complexity and costs. It all sums up a harsh truth: security is not free; on the contrary, it can be very expensive.

It was not until recently, that the rise of cyber attacks and the catastrophic losses that came with them raised security awareness globally. Although the Crowdstrike incident was not a cyber attack, the last few decades have witnessed many impactful attacks that have shaped society's view of cybersecurity. Major examples are the WannaCry ransomware attack [51], which affected more than 200.000 computers in more than 150 countries, or the infamous attack on the Ukranian power grid system [45] which caused a major power outage. The repercussions of security breaches can be drastic both in terms of economic damages and reputation loss [210, 87]. Therefore, service providers and computer manufacturers are actively working on improving their security. Nowadays, both the industry and academia are pouring resources into this field [32], trying to protect vulnerable assets and lucrative targets from all kinds of cyber threats.

Governments have also taken an active role in advancing cybersecurity. Initiatives like the European Union's CrossCon project [66] and the introduction of regulations such as the General Data Protection Regulation (GDPR) [250] have significantly influenced this technology landscape. The GDPR, for instance, sent shockwaves through the web industry by imposing strict data protection requirements. Additional regulations [189, 61] are under discussion, likely driving further research and innovation in this field.

In conclusion, as computer systems continue to pervade every facet of our lives, cybersecurity is no longer optional but essential. While achieving comprehensive security involves substantial costs and challenges, the consequences of neglecting it are far greater. The growing investments by governments, industries, and researchers highlight a collective acknowledgement of cybersecurity's paramount importance in safeguarding

[1] Modern hardware vulnerabilities, such as Spectre [138] and Meltdown [159], illustrate this trade-off. These vulnerabilities, introduced by performance-focused design choices, have given rise to a new class of speculative execution attacks

our interconnected world.

1.2 Trusted Computing, the foundation of security

Security is seldom a binary property: it is hard to declare a system as either secure or insecure. Security is rather an incremental property, with systems being secured *to varying degrees* against specific types of attacks, depending on the assumed attacker model. Despite the great effort that can be invested into the protection of a system, two critical axioms must be acknowledged:

(i) **Attackers adapt and evolve**, continuously finding ways to bypass existing defences;

(ii) **Computer systems are inherently complex**, heterogeneous, and deeply layered, encompassing multiple attack surfaces, each representing a potential target for attackers.

These facts are a major obstacle to security, as they make it hard to predict the effectiveness of a security solution and even harder to establish a system's overall security level.

A clear illustration of axiom (i) is the evolution of control-flow attacks. These are low-level run-time techniques aimed at hijacking the normal execution of a program, such as taking control of its execution flow. While the origins of some attacks are hard to pinpoint, code-injection attacks [191] arguably paved the way for modern approaches like code-reuse attacks, including Return-Oriented Programming (ROP) [223] and Jump-Oriented Programming (JOP) [39]. Each of these techniques represents an evolution of its predecessors, designed to enhance effectiveness or usability or to entirely circumvent existing defences. For instance, when W$\bigoplus$X defences (e.g. Data Execution Prevention (DEP) [171]) were introduced to block the execution of attacker-injected code, ROP shifted the attacks' paradigm by enabling attackers to reuse existing code in a controlled manner to achieve arbitrary code execution. Interestingly, when more advanced defences (e.g. shadow stacks [43]) were appositely designed to prevent this type of attacks, new attack vectors were explored by techniques such as Direct Data Manipulation (DDM) [52, 116], Data-Oriented Programming (DOP) [117] and Block-Oriented Programming (BOP) [126]. Unlike control-flow attacks, these techniques focus on manipulating the program's generic data flow, ultimately enabling attackers to hijack applications. These methods, often referred to as data-oriented attacks, underscore the ever-evolving nature of run-time threats.

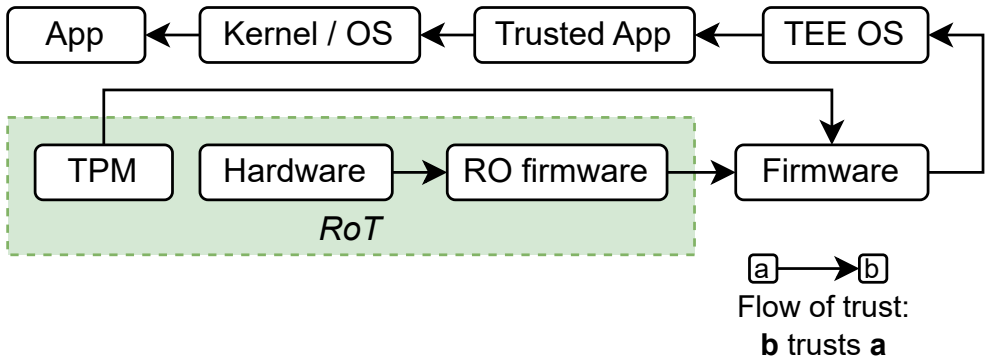


Figure 1.1: Example of a chain of trust in a computer system. A Root of Trust (RoT) is used to establish a set of trustworthy security primitives, which are then leveraged by the next software layers. Each layer usually relies on the previous one, trusting its guarantees. Furthermore, it is common for each layer to verify the integrity of the next one, ensuring that the chain of trust is not broken.

Considering both axioms (i) and (ii), it becomes evident that security is fragmented and uncertain. To address this intrinsic issue, modern computer systems adopt a layered security approach. Each defence mechanism is built atop the guarantees provided by an underlying layer, allowing security modules to be smaller, more focused, and less error-prone. This approach requires a high degree of trust in the underlying components, as their failure compromises the layers above. This concept is known as the chain of trust,[2] an example of which is illustrated in Figure 1.1. Every chain of trust originates from a set of components considered inherently **trustworthy** by the system, i.e. components that rely on no other elements for their security guarantees. This foundational set is commonly referred to as the Root of Trust (RoT), which typically includes a minimal read-only immutable firmware[3], the system's hardware and specialised security components such as the Trusted Platform Module (TPM) or the Trusted Execution Environment (TEE). Interestingly, these advanced components have fundamentally changed how systems build security, implementing separate and hardware-protected environments where critical services can run securely. This paradigm is often referred to as **Trusted Computing**.[4]

Trusted Computing introduces an additional privilege layer beyond the traditional

[2]The term *chain of trust* is also commonly used in the context of SSL certificates.

[3]Not all firmware is considered part of the RoT unless it is stored in read-only memory. More generic firmware is usually verified using techniques such as secure boot.

[4]The terms *Trusted Computing* and *Confidential Computing* are often used interchangeably. While their meanings may vary across different communities, in this thesis, we adhere to the former and use it as defined herein.

dual-layer model of user and kernel spaces. This extra layer is designed to host security services and critical operations, ensuring their functionality and confidentiality even in the presence of an attacker who has compromised the system. Executing such services in the user or kernel layer would expose them to the extensive attack surfaces inherent to those layers, making them vulnerable to exploitation through user- or kernel-level vulnerabilities. For example, if a cryptographic key was stored alongside kernel data, an attacker who breached the kernel could extract it, leading to a complete breach of confidentiality. To mitigate this risk, Trusted Computing isolates the execution and memory of critical services using components such as TPMs and TEEs. These components provide similar security guarantees in many scenarios, though they differ fundamentally in their implementation and in the security services they provide. TPMs are standalone hardware chips dedicated to secure operations, primarily focused on cryptographic operations and secure storage for keys, certificates and measurements. A TPM is usually involved from the earliest stages of the boot process, ensuring firmware integrity and verifying the Operative System (OS) through secure and measured boot. On the other hand, TEEs are hardware-isolated environments that operate within the resources of the System on Chip (SoC) and offer customisable secure enclaves. Compared to TPMs, TEEs are generally more complex, flexible, and versatile. While TPMs have immutable firmware and a fixed set of security services, TEEs can typically run an entire OS while sharing physical resources with the unsecure parts of the system. More importantly, TEEs can be customized to support a wide range of security services, enabling developers to build complex security architectures. However, this increased complexity results in a significantly larger attack surface, often making TEEs more vulnerable to attacks [48]. Additionally, by sharing resources with unsecure system components, TEEs are more exposed to side-channel attacks.

A prominent example of a TEE is Arm's TrustZone [22], which is widely used in Arm-based devices. TrustZone creates a single secure enclave, known as the Secure World, isolated from the untrusted part of the system, the Normal World. The Secure World is physically protected from the Normal World and is responsible for running security services and applications. By extending the CPU buses with an additional Non-Secure bit, TrustZone enables physical isolation of memory and secure peripherals between the two worlds. Although TrustZone introduces only a single security enclave, this enclave supports multiple privilege levels, which are effectively placed beneath (i.e. more privileged than) the traditional Normal World privilege levels. This design allows for the execution of a TEE OS and TEE applications, facilitating the development of complex and independent security services that operate with higher privileges across the system.

To regulate communication and transitions between the two worlds, TrustZone includes a secure monitor component, which is integrated into the system firmware. Intel's SGX [62] is another example of a TEE, designed to protect the confidentiality and integrity of user applications. SGX creates isolated memory regions, called enclaves, where applications can execute securely. These enclaves are protected from the rest of the system, including the kernel and hypervisor, and can only communicate with the outside world through controlled interfaces. While SGX enables multiple user-defined enclaves, each protected from the entire system, it presents challenges when building complex security services and architectures that require interaction between multiple enclaves. This limitation makes it more difficult to implement collaborative security mechanisms compared to other TEEs. In addition to these well-known solutions, many other TEEs exist, each offering distinct features and security guarantees. For example, AMD's SEV [6] and Intel's TDX [9] are well-established TEEs designed to protect virtual machines in cloud environments. Similarly, within the RISC-V ecosystem, security experts are advancing the development and standardization of RISC-V CoVE [213], a fully open-source TEE that enables the execution of confidential virtual machines.

The versatility and high significance of these Trusted Computing RoTs have driven the adoption of TPMs and TEEs as essential components in modern computer systems, to the point that chip manufacturers are continuously innovating, designing advanced versions with enhanced security features. On the one hand, these features enable more sophisticated environments — for example, ARMv9's TrustZone supports multiple distrusting operating systems within the same TEE — which fosters both interoperability and improved security. On the other hand, the increasing complexity of the RoT also extend its attack surface.

In the realm of Trusted Computing, the RoT serves as the cornerstone of security, providing a reliable starting point for securing systems. However, technologies such as TPMs and TEEs are merely a foundational step; their true potential lies in enabling a range of security technologies and services that can protect individual applications or entire systems from diverse threats. Trusted Computing extends beyond traditional cryptographic primitives, offering capabilities to defend against sophisticated run-time attacks, such as ROP, DOP, and other exploit techniques. These pervasive attack vectors can often be mitigated using advanced security mechanisms built upon Trusted Computing primitives.

A broad spectrum of exploit mitigation techniques can be implemented, including, but not limited to, run-time attestation schemes, Control-Flow Integrity (CFI), Data-Flow Integrity (DFI), and various other protective measures. These mechanisms

enhance the system resilience to run-time attackers, by detecting and preventing unauthorised operations. Beyond mitigation, Trusted Computing also facilitates recovery operations, allowing systems to restore secure states following an attack. Additionally, it supports critical security functions such as secure memory erasure, ensuring sensitive data is irrecoverably deleted, and secure code updates, which guarantee the integrity and authenticity of software patches.

By integrating these capabilities, Trusted Computing not only addresses immediate threats but also establishes a robust framework for long-term system reliability and trustworthiness. It plays a pivotal role in defending against evolving attack landscapes while enabling a proactive approach to security.

Interestingly, while these hardware-based security components and techniques are gaining prominence, many systems, especially low-end devices, still lack them. This discrepancy highlights the ongoing challenge of balancing security, cost, and accessibility in an increasingly interconnected world.

1.3 Low-end devices: the forgotten security frontier

Over the past few decades, computer systems have become pervasive, playing a vital role across nearly every sector and aspect of daily life. This ubiquity stems from their versatility and heterogeneity. Today, computers can be so compact and adaptable that they seamlessly integrate into industries, healthcare, agriculture, and smart homes. While society is mostly only aware of those devices we carry with us or use at home, like smartphones and personal computers, billions of smaller, equally sophisticated devices embed a Micro Controller Unit (MCU). These devices, often referred to as low-end, are characterised by limited resources and capabilities, balanced by their low cost, small size and high efficiency. MCUs are almost everywhere and are frequently interconnected with other MCUs or higher-end devices, forming highly complex and densely populated networks. This is exemplified by the Internet of Things (IoT).

Considering the sheer number of these MCUs, their security is of uttermost importance. Although they generally represent low-value targets individually — they generally perform small tasks without holding on to valuable data — these systems are often connected to the internet or they are part of networks with more valuable devices. In the first case, millions of vulnerable devices connected to the internet can be easily exploited to mount global-scale attacks — a prominent example is the Mirai botnet [19] malware, used to orchestrate some of the most significant Distributed Denial of Service (DDoS) attacks to date. In the second case, when a vulnerable device is connected to

a private network, it can serve as a pivot point for attackers targeting other valuable assets within the same network. For instance, in 2018, attackers exploited a casino's fish tank thermometer [262] to gain access to the casino's network.

The density of these heterogeneous networks poses significant cybersecurity challenges. Each device exposes a unique attack surface, and as more devices interconnect, the collective attack surface grows, and the repercussions of a security breach escalate. Notably, the security of a single device might depend on the security of other devices within the network. This is especially true when multiple interconnected devices trust each other implicitly: the less protected devices, i.e. the weakest links, become a potential point of failure for the entire system.[5] Consequently, it is imperative to extend security to every device within a system, including low-end devices.

Unfortunately, while high-end devices typically benefit from robust security measures, MCUs are rarely provided with the same protections. This discrepancy arises from several factors, primarily the bare-metal nature of MCUs. Designed for simplicity and efficiency, these devices often lack kernels or hypervisors, leaving user applications to run directly on the hardware with minimal firmware as an intermediary. Consequently, application developers are often responsible for ensuring security on such devices. By contrast, high-end systems frequently provide out-of-the-box security solutions integrated into their OS or hypervisor. The reliance on application developers for security expertise on low-end devices is problematic and can often lead to unsecure systems. Furthermore, even security-knowledgable developers might struggle to deploy secure systems given the constrained nature of MCUs. Their hardware limitations often mean the absence of useful Trusted Computing components like Trusted Execution Environment (TEE) or a Trusted Platform Module (TPM), which are crucial for establishing common security services (e.g. remote attestation). Worse still, many MCUs lack basic security primitives due to the absence of critical security features such as Memory Management Units (MMUs), Memory Protection Units (MPUs), or even privilege systems — cornerstones of security in higher-end systems.[6]

Interestingly, MCU manufacturers are addressing this security gap by proposing new devices equipped with advanced security features, such as Arm's TrustZone-M [23] and NXP EdgeLock [187]. Notably, TrustZone-M is a simplified version of TrustZone technology, tailored for the real-time operations and memory layout of ARMv8-M MCUs.

[5]Techniques like Zero Trust Architecture [229] aim to minimise attack vectors within device networks. Unfortunately, implementing such principles in real-world complex systems remains challenging.

[6]These features allow the memory in a system to be assigned to different distrusting parties, and to create separated environments where the user applications and the kernel can operate independently and communicate through controlled endpoints.

Unlike traditional TrustZone, TrustZone-M eliminates the secure monitor and introduces a few instructions to enable faster context switching. Additionally, it statically partitions physical memory between the Secure and Normal Worlds. Interestingly, the latest Arm Cortex-R architecture, designed for embedded real-time operations, does not include TrustZone technology. Instead, it relies on an additional hypervisor privilege level to establish isolation primitives. Despite these advancements, the vast majority of deployed devices remain legacy systems that lack such security features.

For these legacy MCUs, there is a scarcity of security solutions, despite their popular usage. One possible reason for this issue is the perception that legacy devices are expendable and easily replaceable with more secure alternatives. However, replacing these devices is neither always feasible nor practical. In industrial sectors, many legacy devices are tightly integrated with the rest of the system and cannot be replaced. In agriculture, the cost of replacing devices may be prohibitive. In consumer markets, the sheer number of devices makes a comprehensive replacement unrealistic. Therefore, it becomes critical to explore security solutions that transcend the limitations of constrained devices. In the absence of adequate hardware security components, carefully designed software can provide essential security guarantees.

1.4 The role of software in security

Modern high-end computer systems often feature feature-rich Roots of Trust (RoT), incorporating hardware components such as TPMs, TEEs, and MMUs. TEEs, in particular, have gained significant popularity in recent years, with various chip manufacturers introducing diverse designs and feature sets. Notable examples include Arm TrustZone [22] and TrustZone-M [23], AMD Secure Encrypted Virtualisation (SEV) [6], Intel Software Guard Extensions (SGX) [62] and Trust Domain Extensions (TDX) [9], and RISC-V CoVE [213]. While TEEs are not bullet-proof, often introducing pernicious vulnerabilities [176], their flexibility and customisability make them pivotal in Trusted Computing.

Although traditional Trusted Computing architectures are built with powerful hardware primitives, their security guarantees are established using several software layers. Beyond the firmware that enables basic interactions with the various hardware components, the software is in charge of managing and orchestrating the various security primitives. For instance, an MMU provides primitives for assigning memory to various entities with various permissions. Still, an MMU requires a kernel or a hypervisor to define and enforce an access control policy, without which the MMU cannot deliver

meaningful security assurances. Similarly, a TEE often only provides the hardware capabilities (accessible through APIs) to create an isolated environment, but it relies on trusted software, commonly referred to as TEE OS, to manage the secure environment and the security services (e.g. the Trusted Applications) executed within.

If, on the one hand, this combination of software and hardware is necessary to have flexible architectures that can meet different requirements, on the other hand, it makes it hard to determine what is the role of software and hardware in such complex systems, where the line that separates the two can be blurred. Interestingly, a higher dependency on software leads to worse performance and wider attack surfaces, while more hardware-dependent solutions lack scalability and versatility (on top of increasing the risk of pernicious hardware flaws). Therefore, it is important to correctly balance software and hardware, creating co-designs that are both performant and flexible. This equilibrium, however, is highly influenced by the hardware capabilities of a device, with low-end systems typically requiring heavier software-based[7] security primitives.

In these low-end devices with limited hardware resources, the role of software becomes critical. Unlike high-end systems, where software builds upon an established RoT, low-end systems must rely on software to establish the RoT itself. On devices that rely on basic security features, such as the Memory Protection Unit (MPU) and the Physical Memory Protection (PMP), several software-based solutions have been proposed. In the emerging RISC-V architecture, Keystone [150] has been introduced as a low-end security framework to enable secure enclaves on an unmodified RISC-V processor. It features a security monitor running in machine mode—the highest privilege level—which manages security enclaves and dynamically configures the PMP. This hardware component, similar to the MPU, is used to isolate each security enclave from untrusted applications. Notably, Keystone does not modify the RISC-V ISA. Similarly, the proprietary Hex-Five MultiZone [199] TEE builds on top of the unmodified ARMv7-M and RISC-V architectures to support multiple secure enclaves, called zones. Each zone is configured at compile-time, defining a set of resources that will be exclusively allocated to it at run-time.

Unfortunately, many legacy devices are equipped with very limited hardware security features and they lack even the basic MPU. Implementing security primitives, such as memory isolation, without traditional hardware support demands the creation of a minimal yet carefully designed Trusted Computing Base (TCB) that emulates many traditional hardware functionalities. This task is fraught with challenges, often

[7] *Software-based* solutions are software-hardware co-designs which primarily leverage software to implement their features.

resulting in suboptimal performance and reduced security guarantees. Nevertheless, it also presents an opportunity to develop fully scalable, cost-effective, and cross-platform security solutions based purely on software components.

Conversely, the very capable high-end systems rely on software for a slightly different purpose. Rather than building the security primitives from scratch, software must manage and administrate them. Furthermore, software can go beyond that and improve the core security guarantees of a hardware component, addressing its limitations and extending its features. This role can be equally important since it can greatly contribute to the security of a high-end device and protect it against advanced attackers.

Software is crucial regardless of a device's hardware, but it must be meticulously designed to fully leverage the hardware and establish robust, efficient security guarantees. While both industry and academia focus heavily on high-end systems, low-end devices are often overlooked, with only a limited number of effective security solutions available for them.

1.5 Problem statement

The high heterogeneity of computer systems and attack vectors makes their security a complex and challenging task. In modern interconnected environments, the security of each individual device is critical, as vulnerabilities in even a single device can compromise the entire network. While high-end systems can often count on feature-rich hardware-based Trusted Computing features, enabling powerful security services that defend against various attack vectors, low-end systems often lack even basic security features. Consequently, protecting low-end devices remains a significant challenge due to their limited resources and capabilities, as well as to the constraints posed by the harsh operational environment they are employed in.

One promising approach to addressing this challenge is the use of software to bridge the gap between these two classes of devices. Software can be used to build effective Trusted Computing architecture, as well as popular security services. These type of software-hardware co-designs can be tailored to the specific needs of individual devices and offer a cost-effective means of improving security. In particular, software can replace the missing hardware features to enhance interoperability, strengthen overall system security, and enable low-end devices to provide similar guarantees and services as their high-end counterparts. On the other hand, software can augment the existing hardware security primitives of high-end devices, extending their security guarantees and mitigating their limitations.

This thesis explores the importance of software-hardware co-designs in the security of computer systems. First, we will focus on software-based Trusted Computing techniques and security services for constrained low-end devices, as a means to provide them with effective security guarantees and advanced exploit mitigation techniques. Then, we will explore the role of software in augmenting the hardware security primitives of higher-end devices, showcasing the importance of software-hardware co-designs across all computer systems.

In particular, this dissertation will try to answer the following research questions:

(**Q1**) *What is the role of software-hardware co-designs in securing low-end devices?*

(**Q2**) *Can software-based Trusted Computing provide a cost-effective and versatile RoT for low-end devices?*

(**Q3**) *Can we build effective software-based security services on top of these devices?*

(**Q4**) *Can software-hardware co-designs further boost the security guarantees provided by hardware primitives on high-end systems?*

1.6 Contributions

In this dissertation, we explore the effectiveness and importance of software-hardware co-designs for building Trusted Computing architectures and services, pushing the security boundaries on low- and high-end devices through novel software-based approaches. Our work highlights the critical role of software in enhancing system security, presenting a comprehensive software-based security stack for resource-constrained devices and a software-augmented exploit mitigation technique for high-end systems.

We begin by analysing the role of security hardware in IoT devices, focusing on a widely used component, the Memory Protection Unit (MPU). This analysis underscores the limitations of hardware in providing robust security guarantees without the proper software support. It reveals the indispensable contribution of software in complementing such hardware components, laying the foundation for our software-based security architectures. This analysis emphasizes the need for solutions like our security stack, which compensates for the absence of dedicated security hardware.

Our security stack comprises PISTIS, a bare-metal Trusted Execution Environment (TEE) specifically designed to provide strong security guarantees in the absence of any security hardware module. Our TEE is the first software-only RoT designed for Von-Neumann based low-end computer systems, with full support to core operations of IoT

devices (e.g. DMA and interrupts) and to the execution of custom security services (a.k.a. Trusted Applications). Through extensive evaluation, we demonstrate the robustness and practicality of PISTIS, showing how it addresses real-world challenges where hardware-based solutions are unavailable or impractical.

To highlight the versatility of our approach, we extend the software-based RoT to other devices and use cases. Notably, we explore its application in intermittent computing, one of the most challenging IoT environments. Here, we propose a checkpoint-based defence mechanism that securely maintains the application state, enabling continued secure operations even in intermittent power conditions. This demonstrates how our techniques can thwart attackers in scenarios where traditional security measures fail, thus highlighting the versatility of software in building security defences.

To further strengthen our security stack, we will introduce a Control-Flow Integrity (CFI) schema, FLASHADOW. This is built as an extension of our security stack, to prevent attackers from corrupting the application's control flow, with a particular focus on defending against Return-Oriented Programming (ROP) attacks. Using a software-based shadow stack, we provide a practical and effective defence mechanism that extends protection beyond the trusted environment.

Finally, we will explore how software can augment existing hardware-based security features, balancing performance with stronger security guarantees. In particular, we will introduce TAGShield, a software-augmented memory tagging schema that overcomes some of the hardware limitations of the emerging Arm Memory Tagging Extension (MTE). TAGShield introduces an effective exploit mitigation mechanism against powerful run-time attackers on high-end devices, showcasing how software can overcome hardware limitations to deliver advanced security guarantees.

1.7 Dissertation outline

The remainder of this dissertation is organised into 6 chapters.

Chapter 2. This chapter introduces the role of security-related hardware in low-end computer systems, examining the challenges of trusting such hardware and the security guarantees it can provide. It includes a case study on the security of the MPU, highlighting its potential shortcomings in delivering robust security guarantees for embedded systems. In this context, we show the importance of software in correctly configuring and managing the hardware, demonstrating how security needs software-hardware co-designs to be effective.

Chapter 3. This chapter lays the foundation of our security stack by introducing the software-based Trusted Computing Base (TCB) as a fundamental building block for security. We describe its design and implementation, demonstrating how software can achieve strong memory isolation, privilege separation, and create a Trusted Execution Environment (TEE) for executing custom Trusted Applications (TAs) on resource-limited bare-metal devices.

Chapter 4. This chapter presents a practical use case of our TEE, showcasing its application in intermittent computing, an application scenario with highly constrained devices. By leveraging the TEE, we introduce a novel security service to address the challenges posed by the intermittent nature of these devices. This service enables secure operation in energy-scarce and unpredictable environments while ensuring protection against attackers and data loss.

Chapter 5. This chapter introduces a novel software-based shadow stack to defend bare-metal devices against control-flow attacks. We demonstrate its effectiveness in protecting devices without hardware security components, highlighting how our TEE can be combined with advanced security defences to achieve a high level of protection for low-end systems.

Chapter 6. This chapter explores an example of software-hardware co-designs on high-end systems, showing how combining software and hardware can significantly enhance system security while maintaining good performance. Focusing on higher-end devices, we go beyond traditional Trusted Computing security defences, introducing an integrity-protected memory tagging scheme to thwart sophisticated attackers.

Chapter 7. The final chapter concludes the dissertation by analysing the real-world impact of our contributions. It explores their integration within the CrossCon project, illustrating their practical applicability and potential to advance the field of security.

Chapter 2

On the (in)security of Memory Protection Units

This chapter is based on a work previously published as:

Grisafi, M., Ammar, M., Crispo, B. (2022, July). On the (in) security of Memory Protection Units: A Cautionary Note. In 2022 IEEE International Conference on Cyber Security and Resilience (CSR) (pp. 157-162). IEEE.

Preamble

In computer systems, hardware must be at the core of every Root of Trust (RoT) because it enables nearly every security primitive. For example, we must rely on the CPU to correctly store the current privilege level in its status registers. While we cannot trust a security primitive without trusting the hardware that underpins it, it is equally unwise to blindly trust a security primitive merely because it is hardware-enabled. In computer systems, software and firmware play crucial roles in managing hardware and must therefore be either integral parts of the RoT or verified by it through techniques such as secure boot.

Interestingly, in high-end systems, firmware is often treated as a black box, bundled up in commercial-off-the-shelf (COTS) devices that are sold as *security-ready* platforms. These are already equipped with an OS verified by the underlying BIOS or UEFI. However, in bare-metal low-end systems, this is hardly the case. On such devices, firmware falls under the direct control of application programmers, who are tasked with configuring the hardware as well as establishing any system-level security primitive. Consequently, on these systems, the security guarantees provided by hardware can

become more fragile.

To demonstrate the crucial role that firmware and software play in low-end systems, we examine the Memory Protection Unit (MPU), an optional component in many embedded devices that can be used to protect the device memory from unwanted access. Unfortunately, the correct configuration of the MPU relies entirely on the application programmer, who has to make security-aware executive decisions to offer the desired level of security guarantees. This task is complicated and error-prone, and any configuration mistake could allow malicious software to completely breach the system, e.g. disabling the MPU. The MPU protection could also be voided by malicious software leveraging specific device features, e.g. interrupts.

In this chapter, we will share our experiences in analysing the security of MPUs and reason on the impact that firmware has on their security guarantees. We first overview the state-of-the-art on MPU security, highlighting the bad practices when configuring the MPU for securing the device memory. We then demonstrate the possibility of exploiting the raised issues, targeting two well-known architectures, namely MSP430 and ARMv7-M. We conclude with a set of recommendations for the correct usage of MPUs.

Chapter outline. The rest of the chapter is organised as follows. Section 2.1 introduces the issue at hand and provides a high-level comparison between MPUs and MMUs. Section 2.2 offers further context by reasoning about the security of low-end devices compared to high-end ones. Section 2.3 delves into the state of security in MPU-based systems. Section 2.5 presents practical attack scenarios on MPU-based embedded systems, while Section 2.6 discusses mitigations to counter such attacks. Section 2.7 summarises related work, and Section 2.8 concludes the chapter.

2.1 Introduction

The Internet of Things (IoT) consists of embedded devices that sense and manage our environment in a growing range of applications. The vast majority of such devices are low-end, lacking rich hardware security features that are available in mid- and high-end systems exemplified by smartphones and laptops. For instance, Trusted Platform Modules (TPMs) and Memory Management Units (MMUs) are usually too expensive to be included in low-end devices, which are instead equipped with less feature-rich components such as Memory Protection Units (MPUs).

Table 2.1: Key differences between MPU and MMU.

	Where	Configured by	Mapping	N. of mappings	App-based isolation	Virtual Memory
MMU	High-end systems	Kernel	Fixed-size pages	Unlimited	Yes	Yes
MPU	Low-end systems	User	Variable-size regions	<= 16	No	No

MPUs are currently supported in many embedded architectures, including Arm, MSP430, and RISC-V. In principle, the MPU is a minimised version of the MMU, providing only memory protection support [112]. When configured by the application programmer, the MPU allows privileged software to define memory regions and assign memory access permissions and attributes to each one. The maximum number of memory regions allowed is architecture-dependent. Despite both targeting memory protection, the security provided by the MPU is quite different from the MMU one. The two modules operate in different contexts and under different threat models, making the understanding of such distinction crucial to avoid security pitfalls. Low-end embedded devices offer a much smaller set of features than their high-end counterpart. Moreover, the programming paradigm of embedded systems revolves entirely around the application programmer, who is in charge of (re-)configuring the system with each application deployment. This paradigm exposes the system to high risks due to the high probability of committing mistakes. In other words, configuring the MPU is a cumbersome and complex operation that can easily introduce bugs and vulnerabilities in the code, widening the attack surface. This is not the case in MMU-based systems, where the underlying kernel is responsible for such configurations.

Several proposals in the literature leverage the MPU to provide security services such as remote attestation, assuming unconditional security guarantees [112, 89, 76]. This is a wrong assumption as such guarantees are only ensured if and only if the MPU is explicitly and correctly configured by the application programmer.

Contribution. In this chapter, we shed light on the security implications of the MPU configuration process and the related security gap, a topic widely overlooked so far. We first systematically evaluate the various security properties of MPUs compared to MMUs, highlighting the pros and cons. We then experimentally discuss several open-source practical attack scenarios, showing the possibility of bypassing MPU-based systems in the absence of correct and rigorous configurations [104]. We conclude with a set of guidelines for a correct and secure usage of MPUs.

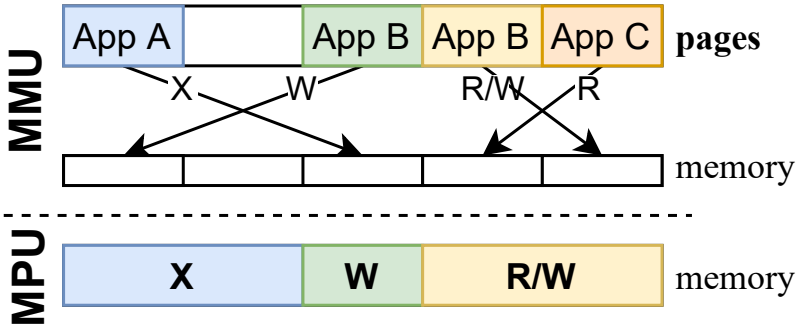


Figure 2.1: Comparison between MMU and MPU memory protection. Single MMU pages are assigned to a single application and virtually mapped to memory, while MPU regions apply directly on physical memory for all running software.

2.2 MPU vs. MMU

The Memory Protection Unit (MPU) is a hardware security feature that was initially introduced to fill a security gap in the low-end Arm processors, providing memory protection support. Since then, the MPU has been implemented in several other architectures and Microcontroller Unit (MCU) families. High profile examples include the MSP430 family, the Infineon MCUs, and the standard RISC-V architecture.[1] Although the key features of the MPU are similar across all families of MCUs, each architecture might use a different implementation, reducing or extending the basic MPU functionality.

The MPU features in terms of memory protection are similar to those offered by the Memory Management Unit (MMU). The MMU, which has been playing a fundamental role in high-end computing systems for a long time, enables features such as efficient memory virtualisation and caching. While the latter is mainly leveraged to speed up memory accesses, memory virtualisation is a pivotal feature that enables an efficient and effective memory isolation, while translating logical addresses into physical ones. In particular, this feature virtually separates the device memory in regions (a.k.a. pages). Each region is then assigned to an application, preventing other applications from accessing it. On top of this isolation, the MMU also provides memory protection by allowing a set of access privileges (R/W/X) to be assigned to each page.

Low-end embedded systems cannot support the MMU. These systems are resource-constrained and usually run a single application. This means that the virtual memory and cache system of the MMU are unnecessarily complex. However, embedded systems

[1] In RISC-V, the MPU is called Physical Memory Protection (PMP) module.

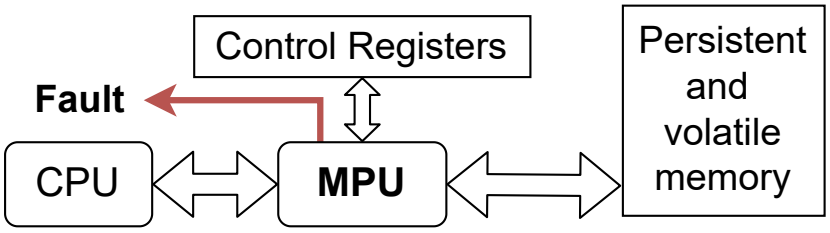


Figure 2.2: Typical hardware layout of an MPU-equipped MCU.

can greatly benefit from memory protection, the reason for which the MPU was introduced in such devices. Practically, the MPU can be seen as a trimmed version of the MMU, offering solely (and also limited) memory protection support.

The MPU allows to restrict memory access over the entire (or part of the) physical memory of the underlying device, splitting it into regions, i.e. variable-length memory areas. The programmer can configure each region and associate it with a custom access policy (R/W/X) enforced by the MPU. This will hold for all applications on the system.[2] Figure 2.1 and Table 2.1 summarise the differences between MPUs and MMUs.

Despite its shortcomings compared to the MMU, the MPU can still be used to build more secure embedded systems. For instance, it can be used to enforce Never-eXecute (NX), an important feature to mitigate code injection attacks. Any violation of the enforced access rights will trigger an exception at the hardware level, which will be handled according to the implementation of the underlying architecture. Figure 2.2 visualises an architecture-independent scheme for the role of the MPU, which is placed between the CPU and the memory, handling all memory accesses on the basis of some control registers (containing the regions configuration). In reality, the MPU is generally offered as an optional feature that can be enabled or disabled at will.

2.3 The state of high- and low-end systems security

Low-end devices have much fewer resources and features at their disposal to keep the cost and power consumption down. Furthermore, bare-metal embedded devices, which form a majority of the embedded world, drastically change what we call the *security paradigm* of a device, i.e. how and by whom security is established and handled. Due to the lack of any kernel, these devices entrust the design, implementation and configuration of any security service to the application programmer. Therefore, MPU-based systems are error-prone, whereas this is not the case in MMU-based systems.

[2]Some MPUs can be reconfigured at run-time to provide flexible policies.

2.3.1 Security in high-end systems

In high-end systems, software is usually separated into kernel and user applications. The former runs always in a privileged mode, i.e. in the *kernel space*, having complete access to the hardware (e.g. the device memory).[3] On the contrary, user applications run with limited privileges, i.e. in the *user space*. The MMU is the hardware component responsible for managing memory privileges, configured by the kernel itself. It allows for a controlled interaction between user applications and the kernel, preventing the violation of the enforced access roles and enhancing the overall security of the underlying system.

Notably, the neat separation between the kernel and user space also exists from the programmer's point of view. In particular, the kernel and user applications are implemented by entirely different parties, referred to as *kernel programmers* and *application programmers* respectively. Following the high-end system security paradigm, application programmers can ignore most of the technical details of the underlying hardware. Moreover, they can blindly rely on the kernel to provide the various required security features.

In this regard, the kernel is a trusted software component, relying on two fundamental assumptions. First, it is created by a highly skilled and security-knowledgeable group of programmers (e.g. for a close-source OS) or by an entire community of experts (e.g. for an open-source OS). Second, the kernel is usually self-contained, preconfigured, securely verified and deployed prior to any other user application. With that in mind, application programmers consider high-end systems secure as long as the MMU is present and managed transparently by the kernel.

2.3.2 Security in low-end systems

In low-end embedded systems, the aforementioned security paradigm substantially changes and the separation between kernel and user applications becomes blurred. Low-end devices are optimised for low cost, small size, and minimal power consumption. Thus, they are neither suitable to support an MMU nor run a monolithic kernel. For instance, the MSP430 family of MCUs only provides a single privilege execution level, meaning that any deployed application has full and unrestricted control over the entire underlying hardware [71].

Practically speaking, in embedded systems, the configuration of both hardware and

[3]For simplicity, we consider a basic system, disregarding technologies such as Intel SGX, Arm TrustZone or AMD SEV that limit the privileges of the kernel.

software is an explicit task of the programmer. This paradigm requires programmers to be highly-skilled in both system and application programming, as well as in the security domain since they are given full control over the system. This has high implications from the security point of view, because programmers can simply fail in properly configuring embedded devices, resulting in vulnerable systems. The presence of an MPU is therefore not a guarantee of the security of the underlying system since it still requires manual intervention to configure it correctly. Moreover, as clarified in Section 2.5, the MPU security guarantees also depend on the rest of the system configuration. Therefore, the MPU is seldom utilised in many embedded systems [271].

This scenario is particularly grave when considering the entire deployment cycle of an application. Programmers often implement an application and deploy it on the device without configuring the MPU. While it is fair to assume that this would not constitute a security problem in high-end systems (due to the intrinsic protection of the underlying kernel), this is not the case in bare-metal devices, where there is no kernel to provide any security guarantee. Even some existing embedded kernels supported in some devices are basic and designed for simple administrative tasks, ignoring any form of security [225]. Therefore, the application would be completely unprotected during its execution. On the other hand, the high heterogeneity of embedded hardware is a major roadblock that hinders finding a unified solution for an automated and error-free MPU configuration process. This exacerbates the security gap in embedded devices by encouraging the deployment of software without the proper security measures.

2.4 Challenges in using the MPU

Designing secure embedded systems is a challenge that requires security-awareness and technical skills, especially considering that there is no user-friendly interface to interact with the underlying fully accessible hardware. A pivotal step is to ensure the correct configuration of the MPU and the rest of the hardware.

Due to their simple architectures, configuring the hardware of MCUs is a minimal yet non-trivial operation. It might involve many intrinsic challenges, especially when considering the MPU. Strictly speaking, even a flawless MPU configuration might not be enough to guarantee the security if the rest of the system is not properly configured as well. For instance, the ARMv7-M architecture of MCUs offers some features that might indeed be used to bypass the protection of a correctly configured MPU. Therefore, the entire system must be designed with security in mind. In general, little attention has been devoted to transparent automated solutions for MPUs configuration [69]. While

such efforts are appreciated, they are hardware- and software-dependent.

Unfortunately, the thorough MCU documentation provided by the manufacturers is seldom comprehensive of all of the security-related risks due to bad configurations of MPUs or other hardware components. Consequently, configuring the MPU according to the simple examples provided in the native guidelines is likely to introduce vulnerabilities. Such vulnerabilities could, for instance, allow an attacker to disable the MPU and thus the memory protection offered by it. It is not surprising that a programmer could overlook these issues and unknowingly build an insecure system.

Notable is that the use of external libraries might exacerbate these security issues. In bare-metal systems, libraries are executed in the same privilege level of the main application, thus capable of compromising the entire system if they are vulnerable. While this problem is well-known and several solutions have been proposed for high-end systems (e.g. privilege separation using compiler assisted techniques such as [35]), embedded systems cannot support practical solutions.

2.4.1 Embedded Operative Systems

Several Embedded Operative Systems (EOSs) have been designed to help the application programmer with the configuration of the MCU or the implementation of complex functionality (e.g. task schedulers). Such EOSs increase the software overhead of the applications, lowering the performance of the MCUs on which they are deployed. Nevertheless, some EOSs (e.g. Mbed OS, FreeRTOS, Zephyr, TockOS and NuttX) offer security features such as memory protection, thus being an appealing choice for embedded application programmers. However, given the high heterogeneity of embedded systems, these EOSs are usually mere Hardware Abstraction Layers (HALs), i.e. a set of libraries that simplifies the interaction with the hardware. HALs aid the programmers in the configuration of the hardware, for instance, by offering them a set of Application Programming Interfaces (APIs) to configure the MPU. Thus, they still require the programmer to make *executive* decisions on the hardware configuration. Even in the case of a correct implementation of the EOS, there might be vulnerabilities in the user application which could lead to compromising the entire system. This is due to the lack of isolation between applications and the corresponding EOS. Furthermore, as clarified in Section 2.5 and as reported in [271], EOSs do not always guarantee the security of the system, even in the presence of an active MPU.[4]

[4]For instance, FreeRTOS has been afflicted by CVEs, such as two integer overflow vulnerabilities: CVE-2021-31571 and CVE-2021-31572.

2.5 A practical overview

As depicted in Figure 2.2, the MPU is a hardware module that can be configured via dedicated, usually memory mapped, registers. Such registers contain information about the MPU state, its regions boundaries, and their access permissions. Moreover, depending on the architecture, these registers might control other secondary features, such as IPE segment in MSP430 or the MPU sub-regions on ARMv7-M.

To configure the MPU, the programmer must write some bits to these registers, following the MCU documentation and some security design rationale. Most manufacturers provide a set of mnemonic variables and functions to facilitate the interaction with such registers, as shown in Listing 2.1. However, as mentioned in the previous sections, the configuration of an MCU and its MPU can be tricky and cumbersome, leading to security vulnerabilities. In what follows, we provide three practical examples of possible attack scenarios. The first two exploit bad MPU and system configurations. The last one is a system workaround. All scenarios are practically demonstrated on MSP430 and ARMv7-M architectures. The source code of all of our demos is publicly available [104].

2.5.1 Bad MPU configuration

The MPU is a component whose configuration requires both practical knowledge of the hardware module and security know-how. Although each MCU manufacturer provides detailed documentation on this component, this usually lacks thorough configuration examples. Moreover, documentation often fails to provide a holistic vision of the security of the system. With that in mind, we believe that a bad MPU configuration is an event that is likely to occur, possibly leading to an attack.

```
MPUCTL0 = MPUPW; //Unlock register
MPUSEGB1 = (0x4500>>4); //Set protection boundary
MPUSEGB2 = (0x10000>>4); //Set protection boundary
//Set privileges
MPUSAM = 0 | MPUSEG1XE | MPUSEG1WE | MPUSEG1RE | MPUSEG2XE |
    MPUSEG2WE | MPUSEG2RE;
//Enable MPU whithout locking it
MPUCTL0 = MPUPW | MPUENA; // | MPULOCK
```

Listing 2.1: Bad configuration of the MPU on MSP430.

To demonstrate how the bad configuration of an MPU can lead to a complete breach

of the system memory protection, we crafted two vulnerable application samples for the MSP430 architecture. In particular, we targeted the MPU-equipped MSP430FR series of MCUs from Texas Instrument. Although the MPU in these devices is not as rich in features as the MPU found on ARMv7-M-based devices, it offers strong security guarantees. For instance, in contrast to the ARMv7-M MPU, the MSP430 MPU can be locked to prevent its modification during run-time. However, failing to lock the MPU can lead to a breach of the system, allowing untrusted portion of the code to freely alter the MPU configuration. Listing 2.1 shows an example of configuration that leaves the MPU unlocked. This configuration, as shown by our *MPU_no_Lock* demo [104], can lead to a complete breach of the system protection, allowing the call to an untrusted library to disable the MPU.

Another possible attacker entry point is the *reset vector*, i.e. a specific location in memory holding the address of the first instruction to be executed after each reset. Since the MPU loses its configuration (i.e. is disabled) after each reset if the configuration code lines are not executed, the attacker can leverage the reset vector to bypass those and thus the MPU. Practically, they can overwrite the reset vector with the address of an instruction following the MPU configuration. As a consequence, at each reset (until the reset vector is changed), the MPU will be disabled. We demonstrate this scenario in the *MPU_reset_Vector* demo [104].

2.5.2 Bad system configuration

The configuration of embedded systems is not trivial for an application programmer. On the more feature-rich MCUs, this can lead to various bad system configurations. In some architectures, e.g. ARMv7-M, the memory protection of the MPU is strictly intertwined with the rest of the system. Therefore, the security is guaranteed exclusively if both the MPU and the rest of the system are configured properly. We provide two attack examples leveraging bad configurations of the bare-metal STM32F207ZG MCU, equipped with an Arm Cortex-M3 processor and an MPU.

The first attack leverages the ARMv7-M privilege system, which offers two distinct execution privileges: privileged and unprivileged. By default, any application runs privileged as long as the privileges are not explicitly lowered. Although the MPU can also limit the permissions of the privileged code, it cannot prevent it from freely configuring the MPU, potentially disabling it. In the demo *MPU_High_Privilege* [104], we correctly set up the MPU but fail at configuring the system properly by not explicitly lowering the execution privileges. To demonstrate a possible attack vector, we crafted

a library that, prior to performing its intended behaviour, disables the MPU, performs some malicious activity, and then re-enables the MPU to go undetected.

In the second example, we do not define an exception handler, i.e. a specific function that is invoked at every system fault (e.g. access violation) to deal with it. Given that in ARMv7-M the exception handlers are executed with full privileges, the code within it has unrestricted access to the MPU registers. It is thus able to disable the MPU or change its configuration regardless of the privileges set by the application. As shown in the *MCU_handler_inject* demo [104], not defining a fault handler allows any third party library to define one instead, effectively taking control of the system by using it to disable the MPU.

It is worth mentioning that our demos contain explicitly malicious code that tries to break the protection set up by the same application, or leverage explicitly untrusted source files posing as external libraries. This scenario is ideal as a proof of concept. A more realistic scenario could be the use of malicious pre-compiled libraries or the presence of code vulnerabilities, e.g. buffer overflows. These could be used to mount Return-Oriented Programming (ROP) or Data-Oriented Programming (DOP) attacks, aiming at disabling the MPU.

2.5.3 System workaround

To mitigate the aforementioned vulnerabilities and establish a sound memory protection, the system needs to be correctly configured. However, in some MCU architectures there can be a few workarounds to circumvent a sound MPU protection. These might be simple features that, despite the system configuration, can pose some threats if the overall security design does not take them into account.

For instance, the ARMv7-M architecture offers a priority-based interrupt system that allows the application code to be interrupted for the execution of specific user-defined routines. However, in this architecture, the interrupt routines are always executed with full privileges without allowing the programmer to explicitly lower them. Interrupts are widely used in embedded systems, yet vulnerable routines might be leveraged by an attacker to alter the MPU configuration and take control of the system. This despite a correct system configuration. We show such scenario in the *MPU_interrupts* demo [104], which correctly configures the MPU and the system, but it also defines an interrupt handler that disables the MPU and performs a malicious operation.

To further demonstrate how the interrupts can breach an apparently secure system and show how an EOS is not necessarily safe, we modified an open-source FreeRTOS-

MPU demo for the Arm Cortex-M3 processor. In the original demo, FreeRTOS initialises two tasks (i.e. two functions), one privileged and one unprivileged, to be executed alternately. FreeRTOS sets up the MPU to protect the system memory from the access of the unprivileged task, effectively preventing the code inside the function from performing unintended operations on the memory. However, this realistic scenario, which is configured with the aid of the FreeRTOS APIs, is vulnerable to malicious interrupts. We implemented an interrupt service routine that disables the FreeRTOS memory protection, allowing the unprivileged task to have unrestricted access to the entire memory.

This proof of concept is particularly relevant considering that FreeRTOS-MPU is supposed to take care of the security of the MCU. However, this EOS is vulnerable to some system-level workarounds that could be introduced by the security-unaware programmers using EOSs as a black box.

2.6 Towards a correct configuration of MPUs

Designing secure systems is a challenge that requires both security know-how and system expertise. We therefore provide some insights (regardless of the different architecture-dependent configurations) on how to correctly configure the MPU in a low-end embedded system, so that this module can be leveraged without introducing vulnerabilities. Our guidelines are as follows:

- The programmer should carefully check the specification of the target MCU, focusing on all properties related to the accompanied MPU module and other system features that could impact this module (e.g. privilege levels, hypervisor calls, Direct Memory Access (DMA) operations).
- Given a re-configurable MPU at run-time, a clear threat model should be derived first for the required system design, highlighting whether such a feature is needed or should be inhibited. Exceptional cases should also be considered, e.g. interrupts or fault handlers.
- The MPU configuration should be executed as early as possible to enforce the required protection level from the early stages of the boot process.
- Since the MPU is disabled at each MCU reset, protecting the *reset vector* should be a priority to prevent malicious code from skipping the MPU configuration when booting.
- The least privilege principle should be considered when designing secure systems

that support run-time modification of the MPU. Untrusted code should be prevented from executing in a privileged mode. In this regard, privileged-only yet untrusted sections of code, e.g. interrupt service routines, should be monitored or sand-boxed, e.g. by incorporating a lightweight Software Fault Isolation (SFI), to avoid causing damage.

- When using EOSs or pre-configured frameworks, the programmer should carefully analyse their instructions, e.g. via a compiler pass, to guarantee controlled interaction and no malicious tampering with the MPU configurations.

We provide some practical examples in [104] on how to fix the configuration of the demos presented in Section 2.5. While some of them are architecture-dependent, the common theme can be generalised and also considered in the design of future MPU-based security services. Following, are some architecture-dependent guidelines.

MSP430 architecture. The vulnerability introduced by the bad configuration in the code snippet of Listing 2.1 can be removed by including the MPU lock. This will prevent any further code from interacting with MPU registers, to either disable them or change their configurations. Moreover, given that the reset of the MCU is a fairly common event that voids the MPU configuration, it is important to ensure that such module is configured after each reset. First, the MPU should be configured at the beginning of the application code (e.g. as the first piece of code being executed). This will prevent any code executed subsequently from breaking the system. Second, the MPU should be configured to prevent overwriting the *reset vector*, by including it in a no-write region.

ARMv7-M architecture. Given the unrestricted access of the ARMv7-M privileged code to MPU registers, it is important to lower the privileges of the application. Listing 2.2 shows the necessary instructions to be inserted after the MPU configuration. Moreover, a fault handler should always be defined, as well as all other handlers, such as the SuperVisor Call (SVC) instruction, that could otherwise be exploited.

Securing the interrupt handlers execution is non-trivial since they are executed always with privileges. One possible solution is the implementation of a wrapper routine. This should return from the native handler, lower the privileges, execute the user routine and then return to the main code. This technique could also be implemented in FreeRTOS, or any other vulnerable EOS. Another solution is implementing SFI only for fault handlers, as demonstrated in [10].

```
/** MPU Configuration ... */
__asm("MOV R0, #01");
__asm("MSR CONTROL, R0");
```

Listing 2.2: Arm assembly instructions to lower privileges, i.e. writing 1 to the CONTROL register.

2.7 Related work

Several proposals have been focusing on supporting memory protection in low-end devices. Only a small fraction of these proposals leverages the MPU to achieve the goal, despite the fact that this module has been introduced for this specific purpose. The main reason for the wide lack of MPU support is the poor design choices of the MPU module, as highlighted in [271], that complicate the manual configuration process and hinder its automation. Furthermore, the MPU, in practice, offers too few regions with insufficient features to set up complex security systems, requiring state-of-the-art solutions to extend this module or find suitable alternatives. A few proposals in the literature introduce software-based memory isolation without leveraging the MPU, e.g. [103], while others have tried to extend the functionality of the MPU, for instance by implementing a custom algorithm for the region assignment to optimise their usage [196], or by proposing an abstraction layer to virtually increase the number of available regions [194].

Although some MPU implementations are more feature-rich than others, such as ones in ARMv8-M and RISC-V architectures, they might still include some limitations dictated by the poor resources of the low-end devices. Furthermore, plenty of low-end devices still rely on rather limited MPUs. Nevertheless, several approaches in the literature leverage this module to implement some security solutions. For instance, SIA [76] is a security architecture that leverages the MPU in MSP430 devices to provide secure checkpointing and remote attestation services for intermittent devices. The work in [112] leverages compiler inserted-code and MPU support to achieve strong application memory isolation.

Despite being an optional component in many MCU architectures, the MPU and its security properties are seldom discussed in the literature. The most relevant example is the work by Zhou et al. [271] that reasons on the various deficiencies of the ARMv7-M MPU, providing an example on how it is insecurely managed by the FreeRTOS OS. However, this work does not investigate many of the implications of using an MPU as we did in this chapter.

2.8 Summary

In this chapter, we examined the advantages and limitations of Memory Protection Units (MPUs), emphasizing their security implications in the design of secure embedded systems. We analyzed the challenges associated with MPU configuration and the potential security risks stemming from misconfigurations. To provide a practical perspective, we studied MPU implementations in two embedded architectures, MSP430 and ARMv7-M, demonstrating best practices for minimizing the attack surface.

Our findings underscore the fragility of low-end device security, even when hardware-based protection mechanisms are present. Furthermore, they provide a first insight on the key role that software plays in securing such devices. We believe this gives an important perspective on the importance of software-hardware co-designs when building secure systems. To further illustrate this need, future work should conduct a systematic review of real-world embedded firmware — both open- and closed-source — following methodologies such as those in [48] and [235], to assess the prevalence of MPU misconfigurations.

In the following chapters, we will try to build on the concept of software-hardware co-design by proposing software-based security solutions that go beyond the mere configuration of the device hardware.

Chapter 3

PISTIS: Trusted computing architecture for low-end embedded systems

This chapter is based on a work previously published as:

Grisafi, M., Ammar, M., Roveri, M., Crispo, B. (2022). PISTIS: Trusted computing architecture for low-end embedded systems. In 31st USENIX Security Symposium (USENIX Security 22) (pp. 3843-3860).

Preamble

The Internet of Things (IoT) encompasses billions of devices that cooperate in diverse environments and across a wide range of applications. Their sheer number and the security-critical scenarios in which they are often deployed (e.g. healthcare, energy industry) underscore the pressing need for robust security measures. This necessity is further highlighted by several past attacks [20, 262] and the increasing number of regulations [189, 61] being introduced to address these vulnerabilities.

While it is clear that low-end devices must be protected alongside high-end systems, their limited resources and constrained hardware make achieving this protection particularly challenging. Recently, several hardware-assisted security architectures have been proposed to mitigate the ever-growing cyber-attacks on Internet-connected devices. However, such proposals are not compatible with a large portion of the already deployed resource-constrained embedded devices due to hardware limitations. Further-

more, such solutions are often architecture-dependent, thus unpractical in heterogenous systems where a general approach to security is crucial to safeguarding as many devices as possible.

To overcome the hardware limitations of low-end devices and still achieve robust security, we propose PISTIS, a software-based Trusted Execution Environment (TEE) that bridges the gap between high- and low-end systems. PISTIS serves as the foundation of our security stack. In the following chapters, we will demonstrate how PISTIS can be leveraged to provide a wide range of security services across various devices. PISTIS enables several security services, such as memory isolation, remote attestation and secure code update, while fully supporting critical features such as Direct Memory Access (DMA) and interrupts. PISTIS targets a wide range of embedded devices including those that lack any hardware protection mechanisms, while only requiring a few kilobytes of Flash memory to store its Root of Trust (RoT) software. The entire architecture of PISTIS is built from the ground up by leveraging memory protection-enabling techniques such as assembly-level code verification and selective software virtualisation. Most importantly, PISTIS achieves strong security guarantees supported by a formally verified design. We implement and evaluate PISTIS on MSP430 architecture, showing a reasonable overhead in terms of run-time, memory footprint, and power consumption.

Chapter outline. The remainder of the chapter is organised as follows. Section 3.1 introduces the problem statement, and Section 3.2 reviews related work to highlight the need for PISTIS. Section 3.3 provides background on the challenges of creating PISTIS. Section 3.4 describes PISTIS in detail, while Section 3.5 outlines the methodology used to verify the design of its memory isolation feature. Implementation details and evaluation results are presented in Section 3.6 and Section 3.7, respectively. Finally, Section 3.8 summarises this contribution and suggests directions for future work.

3.1 Introduction

The last years have witnessed an increased use of embedded devices in many domains, ranging from tiny sensors to industrial control systems. These devices are increasingly connected to the Internet, realising the idea of the Internet of Things (IoT) by connecting digital processes to the physical world. While this connectivity has enabled a vast array of value-added services in all applications, it has also opened the door to a wide variety of cyber-attacks. A high profile example is the Mirai Botnet [20], a clever malware that managed to turn vulnerable IoT devices, mostly specific types of smart

cameras and Digital Video Recorders (DVRs), into zombies to mount the largest DDoS attack to date.

To detect potential attacks while protecting sensitive code and data, various hardware-assisted Trusted Execution Environments (TEEs)[1] have been proposed by both industry and academia, for high-end (e.g. Trusted Platform Modules (TPMs) [248], Intel SGX [62], and AMD SEV [129]), mid-range (e.g. Arm TrustZone [22]), and low-end platforms (e.g. Arm TrustZone-M [23], VRASED [185], SANCUS [183], and TyTan [41]). All of these TEEs provide confidentiality and integrity guarantees. The former is provided by enforcing strict access control on part of the memory through rich hardware features or some extensions to the original processor architecture, while the latter is realised by some sort of malware detection mechanisms, mainly remote attestation ($\mathcal{RA}$), that is supported on all architectures. $\mathcal{RA}$ is a security service that detects malware presence on a remote, potentially compromised, device, called prover, by verifying its software integrity using an integrity-ensuring function (e.g. a hash-based Message Authentiation Code (MAC)) by a trusted party, called verifier.

Problem statement. Embedded devices are characterised according to many factors, among which, the support of hardware security features. In contrast to mid-range and high-end devices, low-end embedded devices are optimised for low cost, small size, and low power consumption. This makes them incapable of defending themselves against malware infestation attacks. In particular, a significant number of low-end embedded devices depend on commercial-off-the-shelf (COTS) Microcontroller Units (MCUs), which even lack basic hardware-based memory safety features such as Memory Protection Units (MPUs), offering no (or, at best, little) security guarantees. Furthermore, the software in such devices runs bare-metal with no Operating System (OS) support. High profile examples include all AVR ATmega MCUs, many of the Arm Cortex-M family,[2] and all Flash-based MSP430 MCUs. Notable is that the underlying memory architecture of such MCUs is either the Von Neumann one [257], in which both program code and data reside in a single address space, or the Harvard architecture [228], in which code and data memories are physically separated.

Initially, several software-based $\mathcal{RA}$ protocols were proposed to detect malware presence on low-end devices [220, 153, 221]. Such protocols are hardware-independent. They rather rely on assumptions which, however, are hard to achieve in practice, such

[1]Please note that we use the terms "Trusted Execution Environment" and "Trusted Computing Architecture" interchangeably.

[2]Please note that Arm Cortex-M0 has by default no MPU, whereas the MPU is optional in other ARMv7-M-based processors of the same family.

as silent adversary during $\mathcal{RA}$, optimal code, and one-hop communication [27]. This makes them vulnerable to several attacks [46], providing no certain security guarantees. More recently, various hardware-assisted $\mathcal{RA}$ mechanisms with different assumptions and requirements have been proposed for embedded devices [183, 185, 83, 139, 41]. While such proposals provide strong security guarantees, their future availability in low-end devices is questionable due to several reasons. First, these proposals are not compatible with legacy MCUs as they require hardware modifications and thus replacing millions of the already deployed sensors and actuators. Second, changing vendor production lines as a result of hardware modification is not always easy and incurs extra costs. Third, pure hardware solutions, such as SANCUS [183], are inflexible in terms of patches in case of a vulnerability, requiring changing all affected platforms. As an example, a group of researchers exploited an unpatchable hardware design flaw in Xilinx 7-Series FPGA boards which led to a full break of the bitstream encryption, resulting in the total loss of authenticity and confidentiality [85]. Another example includes the removal of Intel Memory Protection Extension (MPX) from all future Intel processors due to several design flaws [198].

Despite the limited initiatives to propose reliable software-based security services for low-end devices [14, 16, 145], existing proposals are limited to the Harvard architecture, which is simpler than the Von Neumann one. Furthermore, the security in such proposals is guaranteed under the assumption that the underlying embedded device lacks (or physically disables) Direct Memory Access (DMA), which is an important functionality that cannot be omitted in many low-end devices. Last, security services in such proposals execute atomically by disabling all interrupts, thus negatively influencing the correct functioning of safety-critical applications.

Contributions. This chapter tackles the aforementioned issues by proposing PISTIS,[3] a pure-software Trusted Computing architecture for low-end Von Neumann-based embedded devices. PISTIS is built from the ground up, targeting embedded devices with limited or no hardware security features. It only requires a few kilobytes of the Flash memory of the corresponding device to hold its Trusted Computing Module (TCM). The TCM is a set of software functions that acts as a hypervisor, enabling different security services, such as memory protection, remote attestation, and secure code update. PISTIS provides strong security guarantees, comparable to those in hardware-assisted architectures, while securely supporting DMA and interrupts. Furthermore, PISTIS has the advantage of being portable to several MCU architectures because

[3]PISTIS was the personification of *good faith* in Greek mythology.

Table 3.1: PISTIS vs. state-of-the-art Trusted Computing architectures from various perspectives.

Architecture	HW modification	Memory Organisation	DMA support	Interrupts support	Formally Verified	Extensibility*
SANCUS [183]	Yes ✗	Von Neumann	No ✗	No ✗	No ✗	No ✗
VRASED [185]	Yes ✗	Von Neumann	Yes ✓	No ✗	Impl. △ Design ▲	No ✗
SMART [83]	Yes ✗	Von Neumann	No ✗	No ✗	No ✗	No ✗
TrustLite [139]	Yes ✗	Harvard	No ✗	Yes ✓	No ✗	No ✗
TyTAN [41]	Yes ✗	Harvard	No ✗	Yes ✓	No ✗	No ✗
SμV [14]	No ✓	Harvard	No ✗	No ✗	Implementation △	Yes ✓
PISTIS	No ✓	Von Neumann	Yes ✓	Yes ✓	Design ▲	Yes ✓

*Extending or updating existing hardware-assisted architectures can only be considered for future devices

of its nature as a pure-software solution. In addition to the formally verified design, PISTIS is accompanied by an open-source implementation for the MSP430 family of MCUs [105].

In a nutshell, this chapter makes the following contributions:

1. **PISTIS:** To the best of our knowledge, this is the first design of a full-featured software-based Trusted Computing architecture that offers memory isolation, remote attestation, and secure code update services with strong security guarantees, targeting Von Neumann-based embedded devices.

2. **Formal Verification:** We formally verify the correctness of the memory isolation design that provides the foundation for establishing the Root of Trust (RoT) in pure software.

3. **Open Source Release:** We release PISTIS as an open-source C library along with a GCC compiler plugin (via [105]) with APIs that automate the deployment of PISTIS as well as the generation of PISTIS-compliant binary images.

4. **Extensive Evaluation:** We evaluate PISTIS covering all relevant performance metrics, including run-time overhead, memory footprint, and power consumption, using a representative set of 13 applications.

3.2 Related work

The constant threat of cyber-attacks on computing systems has driven the design of numerous security technologies, among which TEEs are one of the most widely proposed in the literature and most employed in diverse computing platforms. Existing TEEs generally depend on pure hardware or leverage a combination of software and hardware modules to protect one or multiple running applications by providing them with confidentiality and integrity guarantees.

In high-end (and some mid-range) platforms, various hardware-based TEEs have been designed and implemented for different processors, including Intel, Arm, and AMD. For instance, Intel Software Guard Extension (SGX) [62] extends the Instruction Set Architecture (ISA) of Intel processors to create hardware-enforced virtual containers, called *enclaves*, capable of protecting the integrity, confidentiality, and run-time state of internal applications without trusting any existing software module, e.g. the OS. Arm TrustZone [22] is a well-known TEE for Arm processors, which provides a single hardware-based enclave, dividing the memory into two zones: secure and insecure. Thus, all sensitive apps run in a physically isolated secure memory area. Generally, existing solutions for high-end platforms depend on complex and expensive hardware features that cannot be supported on low-end devices due to cost and size constraints.

Given the lack of rich hardware features on low-end embedded platforms, various types of lightweight security architectures have been proposed. Initially, the main focus was on software-based solutions that enable security services such as remote attestation and secure code update without depending on hardware [221, 220, 222, 153]. In particular, such proposals depend on precise time measurements where the upper bound is estimated during the initialisation of the embedded device (assuming time-space optimal code), or require filling the free memory space with true randomness in order to prevent malware from using it. As aforementioned, such solutions are vulnerable to some attacks as demonstrated in [46]. Furthermore, their security guarantees are uncertain due to depending on unrealistic assumptions, e.g. passive remote adversary, optimal code, one-hop communication, etc.

In contrast to software-based solutions, SANCUS [183] has been proposed as a lightweight security architecture for embedded devices that implements a pure-hardware trust anchor. Compared to other hardware-based TEEs, e.g. SGX [62], SANCUS is cheaper and more suitable for embedded devices. However, it still requires significant changes to the underlying architecture of these devices, increasing their cost. To overcome this limitation, several hybrid architectures have been introduced, depending on a software-hardware co-design [185, 83, 139, 41]. While there are several trade-offs between the designs of these hybrid approaches, they aim to provide the same security guarantees as hardware-based ones, while minimising modifications to the underlying hardware. Although these approaches yield strong security guarantees and good performance on low-end IoT devices, they require customised hardware support on every device, which is either absent or too expensive to implement in certain scenarios. For instance, a wide spectrum of embedded devices that lack any hardware support is already employed in privacy-sensitive and safety-critical domains, making the adoption

of such hybrid architectures impractical [169, 224]. Notable is that VRASED [185] is the only formally-verified hybrid architecture that provides a secure and sound remote attestation ($\mathcal{RA}$) service.

The security MicroVisor (SμV) [14] filled the sizeable gap between software-based and hardware-assisted security architectures by implementing a software-based memory isolation technique to isolate its RoT software (TCM) from the rest of other untrusted software modules running in the same address space. To do so, the SμV is inspired by the Software Fault Isolation (SFI) approach proposed by Wahbe et al. [259]. However, in contrast to all existing memory protection techniques that reuses SFI in high-end computing systems [234], the SμV does not trust the compiler toolchain, relying only on its TCM. Furthermore, it targets low-end Harvard-based MCUs. On top of memory isolation, the SμV provides various security services, such as remote attestation (e.g. SIMPLE [16]), secure erasure (e.g. SPEED [17]), and secure code recovery (e.g. VERIFY&REVIVE [12]). Nevertheless, the SμV does not support either DMAs or interrupts in the aforementioned security services. Notable is that the SμV's implementation (and not its design) has been formally verified w.r.t. to safety properties, targeting only the Harvard-based AVR architecture.

Although PISTIS shares similarities with SμV in terms of adapting and optimising some of the SFI techniques in a new setting, it targets the Von Neumann architecture, which is more challenging than the Harvard one as clarified in Section 3.3.2. Furthermore, it aims for supporting DMA and interrupt operations with critical security services.

To sum up, Table 3.1 compares PISTIS with the state-of-the-art approaches.

3.2.1 Alternative solutions

In the previous paragraphs, we explored the state-of-the-art in TEE-like technologies targeting low-end devices. These approaches operate without relying on platform-specific hardware, such as a Memory Protection Unit (MPU), making them suitable for architectures like MSP430 and AVR. However, several other works have proposed TEE-like security mechanisms and other security techniques for low-end devices with more advanced hardware capabilities.

Compartmentalization

Compartmentalization has been widely adopted as an alternative approach to reducing a system's attack surface. This technique segments software into isolated modules and

enforces memory and privilege separation at run-time. While it does not enable security services to run in a separate privileged execution environment as TEEs do, it enhances security by restricting each module's privileges, thereby limiting the impact of potential vulnerabilities.

Several compartmentalization techniques [31, 58, 135, 272, 134, 170] have been proposed for more capable architectures, such as ARMv7-M, which includes an MPU. These techniques typically use compiler extensions to partition embedded firmware and RTOS components into isolated modules. A lightweight run-time system then dynamically configures the MPU to enforce access control policies based on the currently executing module. By carefully restricting access to system resources — including peripherals, memory regions, and global variables — compartmentalization techniques significantly reduce the overall attack surface of embedded systems. Some approaches further optimize MPU configuration to minimize overhead, ensuring compatibility with real-time constraints. While these methods differ from TEE-based protections, they provide a practical means of enhancing security, particularly in embedded environments where hardware-based isolation mechanisms are limited or unavailable.

Hypervisor-Like Approaches

Rather than splitting a single software system into isolated modules, some approaches adopt a hypervisor-like model to isolate different software components within the system.

Hermes [137] introduces virtualization for ARMv7-M to reduce I/O latency and enable more deterministic execution. It consists of a monolithic exception handler that intercepts all interrupts and dispatches them to the appropriate component. Every software module, including RTOSes, executes in unprivileged mode, with Hermes trapping and emulating all privileged operations. While not originally designed for security, this technique can also be used to execute isolated applications securely.

MultiZone [199], in contrast, focuses explicitly on security, enabling multiple equally secure functional domains (or zones) with restricted access to system resources. It enforces strict compartmentalization via a lightweight separation kernel, the MultiZone run-time, which adheres to zero-trust principles by isolating critical components such as real-time tasks, network interfaces, and cryptographic functions. Secure inter-zone communication is achieved through a message-based mechanism, while ARMv7-M privilege levels and the MPU enforce access control based on user-defined policies.

TEE Implementations for RISC-V MCUs

Although most prior work targets ARMv7-M devices, recent efforts have extended security mechanisms to the emerging RISC-V architecture. Unlike ARMv7-M, RISC-V is highly configurable, with chipsets varying significantly in hardware capabilities based on their use case. The RISC-V ecosystem defines two primary hardware profiles: the **Application Profile** for high-end processors and the **MCU Profile** for embedded devices. While much of the literature focuses on TEEs for the Application Profile, some research has explored secure execution for RISC-V MCUs.

Timber-V [263] extends the RISC-V hardware specification to meet the security requirements of a TEE. However, software-based solutions have also been proposed for standard RISC-V MCU implementations, which feature hardware capabilities similar to ARMv7-M. MultiZone TEE [199] was originally designed for RISC-V, offering comparable features to its ARMv7-M counterpart.

Keystone [150] is another popular TEE for RISC-V MCUs, supporting multiple secure enclaves managed by a lightweight Security Monitor. This low-level software component dynamically configures the Physical Memory Protection (PMP) to enforce memory isolation and access control. Secure inter-enclave communication is achieved through shared memory and a mailbox mechanism. Keystone also integrates optional security enhancements, including cache partitioning for side-channel protection, on-chip memory for resilience against physical attacks, and dynamic memory resizing to accommodate varying workloads.

Finally, Penglai [88], a widely used open-source TEE for high-end RISC-V processors, recently introduced a closed-source variant for RISC-V MCUs.

Leveraging TrustZone-M

While software-based approaches are particularly valuable in the absence of hardware-based Trusted Computing technologies, they can also complement or enhance existing solutions such as TrustZone-M. This TEE technology, present on some of the newer ARMv8-M chipsets, introduces hardware primitives that enforce isolation between the Secure and Normal Worlds. However, several works in the literature have built upon TrustZone-M to extend its capabilities.

Some approaches leverage TrustZone-M to compensate for the lack of a hypervisor-level execution mode. Pinto et al. [200] propose a hypervisor that manages two isolated Virtual Machines (VMs), one running in the Secure World and the other in the Normal World. The hypervisor itself operates within TrustZone-M, leveraging the Security

Attribution Unit (SAU) to enforce spatial and temporal separation between the VMs.

Similarly, SBI [195] introduces a hypervisor designed for zero-software-cost interrupt delivery. It routes all interrupts directly to user-level, bypassing the kernel. Although SBI is compatible with multiple VMs, its TrustZone-M-based implementation supports only one VM per core.

Other works use TrustZone-M to construct security-oriented TEEs. RT-TEE [261] is a real-time TEE designed for Cyber-Physical Systems, ensuring system availability in safety-critical environments. Specifically, it guarantees timely access to system resources (CPU and I/O) for critical processes residing in the Non-Secure World.

SafeTEE [215] targets safety-critical multi-core systems, isolating security-critical applications from safety-critical ones. Each application is assigned an exclusive core, which may or may not have access to the TEE's services, depending on real-time requirements. SafeTEE enforces strict access control policies using the SAU, restricting memory and peripheral access.

Finally, uTango [190] is the first multi-world TEE for TrustZone-M-enabled IoT devices. It introduces multiple equally secure execution environments within the Normal World, providing fine-grained security separation. Following the principles of minimal implementation, least privilege, and containment, uTango dynamically reconfigures the SAU to enforce strong compartmentalization while supporting flexible execution of diverse workloads, including RTOS-based applications and security-critical services. This novel multi-world approach enhances IoT device security without requiring specialized hardware modifications.

3.3 Preliminaries

3.3.1 Scope of embedded devices

PISTIS targets tiny embedded devices that have small MCUs based on the Von Neumann architecture with little or no hardware security features. In general, such MCUs have a single core and feature Flash and SRAM memories. They execute instructions in place (in physical memory) and have no Memory Management Unit (MMU) to support virtual memory. Some of them can support an MPU. However, our design and implementation neglect the existence of MPUs due to their shortcomings as clarified in [271]. In particular, PISTIS targets single-thread, yet multi-tasking bare-metal applications that are the most common ones in the IoT domain [59].

Our implementation is based on the MSP430 architecture. This choice is due to

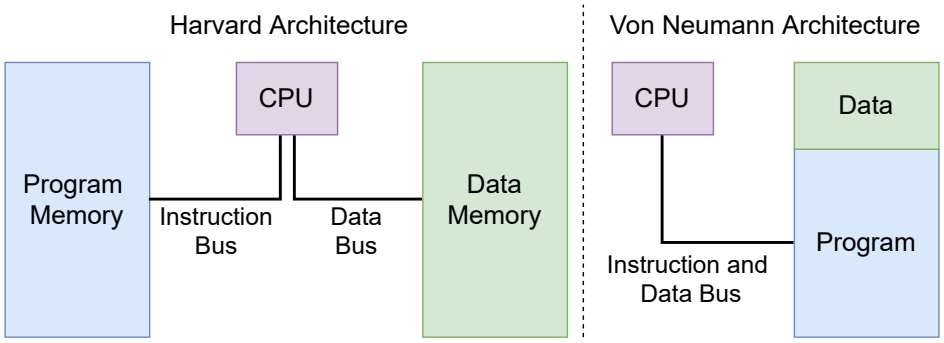


Figure 3.1: Harvard vs. Von Neumann architecture.

the wide use of this architecture in many IoT devices as well as research prototypes. Nevertheless, our design is applicable to other low-end MCUs in the same class, such as Arm Cortex-M.

3.3.2 Challenges in designing PISTIS

All MCUs used in small embedded devices (regardless of their vendors) are designed based on either of two memory architectures: Von Neumann or Harvard. In contrast to the SμV [14] that is proposed for the Harvard architecture, PISTIS targets the Von Neumann one. Both architectures are visualised in Figure 3.1. There are two main challenges in the Von Neumann architecture compared to the Harvard one. First, the Harvard architecture features a hardware isolation address space as both program (non-volatile Flash) and data (volatile SRAM) memories are physically separated, having different instructions for accessing them. This feature simplifies isolating part of the memory to host the trust anchor as the data memory is not executable, and thus no need to instrument its related instructions. This is not the case in the Von Neumann architecture where both program and data memories share the same memory address space and thus are executable, facilitating code injection attacks as basically any instruction could alter the state of the program memory, i.e. read from or write to it. This poses a challenge when designing PISTIS without incurring an intolerable performance overhead. Second, in contrast to the Harvard architecture, the Von Neumann one has a variable-length instruction set that would be exploited to break any software-based memory isolation technique by jumping into the middle of a multi-word instruction and executing one of its words as an unaligned instruction. This challenge cannot be handled by borrowing some of the well-known SFI techniques, i.e. as in the SμV [14], without considering a careful design to adapt them. To solve the first challenge while avoiding adding a high performance overhead, PISTIS focuses on checking the posi-

tion of the Program Counter (PC) rather than instrumenting the entire instruction set. Also, PISTIS devises a special canary mechanism to mark legitimate jump targets, thus handling the second challenge. Furthermore, PISTIS, compared to all existing architectures regardless of their types, is the only solution that supports both DMA and interrupts while executing critical operations. Further details are provided in the next section.

3.4 PISTIS

Overview. Systems security strongly relies on the concept of *trust* as lacking it renders any security service infeasible. Trust is typically assured via some sort of *memory isolation*, a mandatory security primitive for all security services. To this end, PISTIS follows the bottom-up approach to build a pure-software Trusted Computing architecture that is highly optimised for low-end embedded devices with a full support for DMA and interrupt operations. To achieve so, PISTIS initially deploys an initial code, called a Trusted Computing Module (TCM), that occupies part of the Flash memory including the bootloader area. The main goal of the TCM is to guarantee memory protection by isolating its memory area from other memory parts, creating two logically isolated memory zones: secure and insecure. PISTIS leverages the secure memory part to deploy two exemplar security services, namely remote attestation and secure code update. Please note that while PISTIS adapts some of the SFI techniques, i.e. binary re-writing, in its special domain, it does not depend on any hardware feature, such as segmentation registers in MPUs as in [112]. Furthermore, in contrast to standard SFI mechanisms [234], PISTIS represents a full-fledged Trusted Computing architecture, offering a run-time virtualisation environment without trusting the compiler toolchain. The correctness of the memory isolation design in PISTIS is formally verified as clarified in Section 3.5.

In what follows, we outline our adversary model and then describe the design of PISTIS in detail.

3.4.1 Adversary model

We consider a software-based adversary $\mathcal{A}\mathsf{dv}$ who has full access to the network or potentially presents inside the device itself in the form of malware. $\mathcal{A}\mathsf{dv}$ can eavesdrop on or tamper with traffic on any communication medium supported by the target device. However, she cannot launch Denial of Service (DoS) attacks. Protection against DoS

attacks is out-of-scope as they do not tamper with the device memory. $\mathcal{A}\mathsf{dv}$ can also control any software deployed in the insecure memory part of the device. This includes trying to inject malicious code, reading or writing any memory address that is not explicitly protected, corrupting specific data, or manipulating I/O pins.

We assume that the TCM is initially and correctly installed on the embedded device by a trusted party. We also assume that the TCM is bug-free and does not contain any memory corruption vulnerabilities.[4] This means that $\mathcal{A}\mathsf{dv}$ cannot bypass any protection rules enforced by the TCM.

We rule out all physical and hardware-focused attacks. We consider that protection against physical attacks is an orthogonal problem and can be achieved, for instance, by deploying the device inside a tamper-proof protection shield [209].

3.4.2 PISTIS: from the ground up

In what follows, we will describe the main building blocks of PISTIS, namely the TCM and the accompanied compiler toolchain, that guarantee memory isolation, a key-enabling property for all Trusted Computing architectures. We then extend the described design by enriching the TCM with some security services, called Trusted Applications (TAs), i.e. remote attestation and secure code update, leveraging memory protection as a basis. All building blocks sum up PISTIS.

Memory isolation: design rationale

The memory isolation property of PISTIS aims at enforcing memory protection to guarantee the integrity and confidentiality of PISTIS's TCM and its TAs. This is achieved by first deploying the TCM on the device using a physical programming device, e.g. JTAG, by a trusted party. The TCM is responsible for protecting itself against any untrusted software that is deployed on the same device afterward. To achieve so, the TCM requires any software to be deployed through it to verify its safety at the instruction level. The entire deployment will be rejected by the TCM if there is at least one unsafe instruction that violates the memory isolation property. PISTIS is accompanied by a PISTIS-enabled compiler toolchain that produces compatible binary images. However, this toolchain is untrusted as it is easy for $\mathcal{A}\mathsf{dv}$ to tamper with it. Therefore, the security of PISTIS depends on the load- and run-time verification that occurs on the device itself by the TCM.

[4] Likewise the work in [14], the memory-safety property can be achieved by formally verifying the code before deploying it.

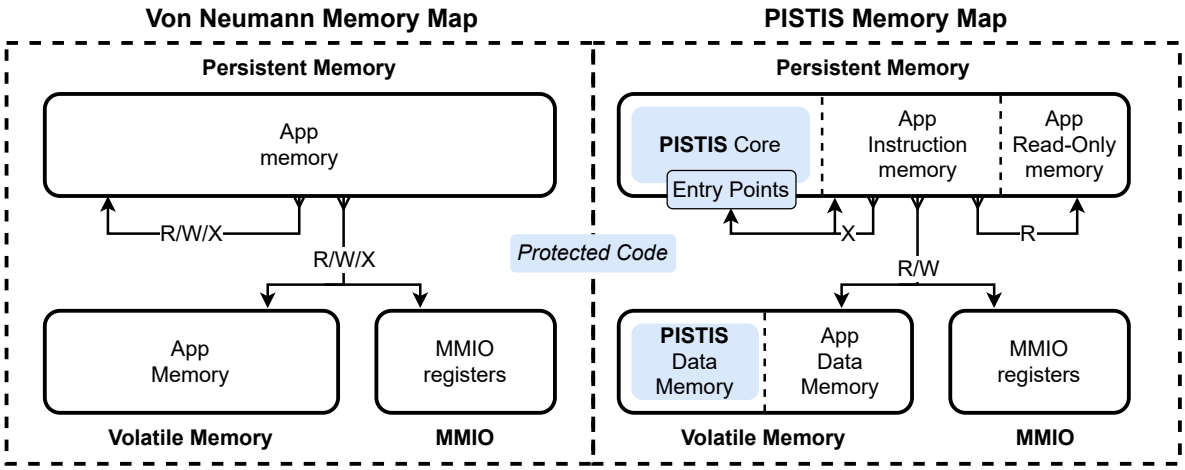


Figure 3.2: A standard memory map vs. the PISTIS-enabled memory map in a Von Neumann architecture. The latter also shows R/W/X privileges of the untrusted application.

Trusted Computing Module (TCM). The TCM is a set of software functions that reserves part of the non-volatile memory, including the bootloader area, to act as a hypervisor that fully manages access control to the entire memory area. In particular, the TCM consists of the following components:

- **Initial code**: a bootloader code that replaces the original one. When the MCU is powered on, this code decides whether to continue booting from the TCM memory or from the memory that holds untrusted software.
- **Loader/Verifier**: this software module is responsible for receiving the untrusted software image from the network interface and verifying whether it is PISTIS-compliant. If so, the software will be installed and activated on the device. Otherwise, it will be rejected and erased.
- **Virtualised Instructions**: a set of functions that represent safe equivalent versions to some potentially unsafe instructions that cannot be checked at load-time. During compilation, the PISTIS-enabled toolchain will replace each instruction with a hyper-call to a safe equivalent one that can be safely verified at run-time.
- **Helper modules**: other necessary helper functions such as the ones that read and write memory pages.

The TCM code runs in a privileged mode in the sense that it can perform any memory access operation. On the contrary, the untrusted software is subject to some restrictions that limit its access to some memory regions. Such restrictions are meant to enforce memory protection and are thus checked and enforced by the TCM. Figure 3.2

shows the memory layout of the MCU when activating PISTIS. The TCM is the first to be physically deployed in the *PISTIS Core* memory part. It then manages other deployments to maintain the shown layout. When deploying an application, the TCM verifies its instructions according to the following Access Policy (AP):

- Read access is limited to Memory Mapped Input/Output (MMIO), Application Data and Application Read-Only memory.
- Write access is limited to MMIO memory[5] and Application Data memory.
- Jumps (to execute instructions) are limited to Application Instruction memory and specific entry points of the PISTIS Core memory.

In contrast to the Harvard architecture where maintaining memory isolation only requires taking care of control-transfer instructions, any instruction in the Von Neumann architecture can violate the aforementioned AP. Therefore, to adhere to such a policy, the TCM's Loader/Verifier should smartly verify each memory access (read/write) or control-transfer (jump/call) instruction of untrusted software before deploying it on the device. Initially, the TCM only knows the boundary of its non-volatile memory area and the dedicated space in the volatile one. To know the other boundaries shown in Figure 3.2, the PISTIS-enabled compiler toolchain produces some meta-data that is sent ahead of the deployed binary image. The TCM performs the verification process according to such meta-data. Please note that the values of such meta-data are accepted as long as they do not cross protected memory regions. With that in mind, the TCM installs the entire received binary image in the expected part of the non-volatile application memory. It then starts verifying it after disabling all interrupts to ensure atomicity. The following checks must be passed before the actual deployment of untrusted software:

- Instructions with a static addressing mode, whose target memory address is known, are checked and verified at load-time. These instructions can be either control-transfer or memory-access ones. They both must comply with the AP visualised in Figure 3.2 in the sense that:
  - control-transfer instructions can only target the memory area where the application will be installed or one of the entry points of the TCM. This check guarantees the Data Execution Prevention (DEP) property of the data memory since PC will not be allowed to jump there.

[5]Exceptions can be made for critical MMIO registers.

  - Write instructions can only target the permitted part of the volatile data memory (RAM) or (some) MMIO registers.
  - Read instructions can only target any write-permitted memory location or the application read-only memory.

- Instructions with a dynamic addressing mode whose target address is only known at run-time are replaced with static instructions in the form of hyper-calls to their secure virtualised versions (part of the TCM). If, at run-time, the target address is deemed to be unsafe, PISTIS performs a soft reset of the MCU to block this operation.

If the binary image contains at least one instruction that does not pass the above checks, it will not be deployed and accordingly erased. The list of instrumented instructions is described in Table A.1 in Appendix A.1. Please note that PISTIS does not support self-modifying code in the sense that the untrusted software cannot directly write to its instruction memory area. However, if required, the untrusted software can invoke the TCM's Loader/Verifier module after installing the needed chunks of code in the non-executable data memory. The TCM's Loader/Verifier will relocate the installed chunks to the required memory location if they adhere to the aforementioned AP.

Modified toolchain. PISTIS leverages a modified toolchain to transparently re-write each potentially unsafe dynamic instruction, and replace it with a safe virtualised equivalent that can be accessed via a call to a subroutine stored in the protected TCM memory area. The target address of the corresponding instruction is verified at run-time when the subroutine is invoked. The execution continues normally if it is valid. Otherwise, an MCU reset is triggered.

Figure 3.3 visualises the modified compiler toolchain. We chose the open-source GCC compiler toolchain and modified it to suit our needs. Our modifications represent: (i) an instrumenter plugin that verifies the validity of static instructions and re-writes dynamic ones, and (ii) a custom linker script that maintains the required memory layout and resolves the addresses of valid TCM's entry points. The instrumenter module is placed between the compiler and assembler, targeting assembly instructions. During instrumentation, all instructions with a static addressing mode are left untouched as they are checked at load-time by the TCM on the device itself. All control-transfer and memory-access instructions with a dynamic addressing mode are re-written as previously clarified. Please note that $\mathcal{A}\mathsf{dv}$ cannot write hand-crafted assembly instructions or use her own toolchain without being detected by the TCM. Application developers can

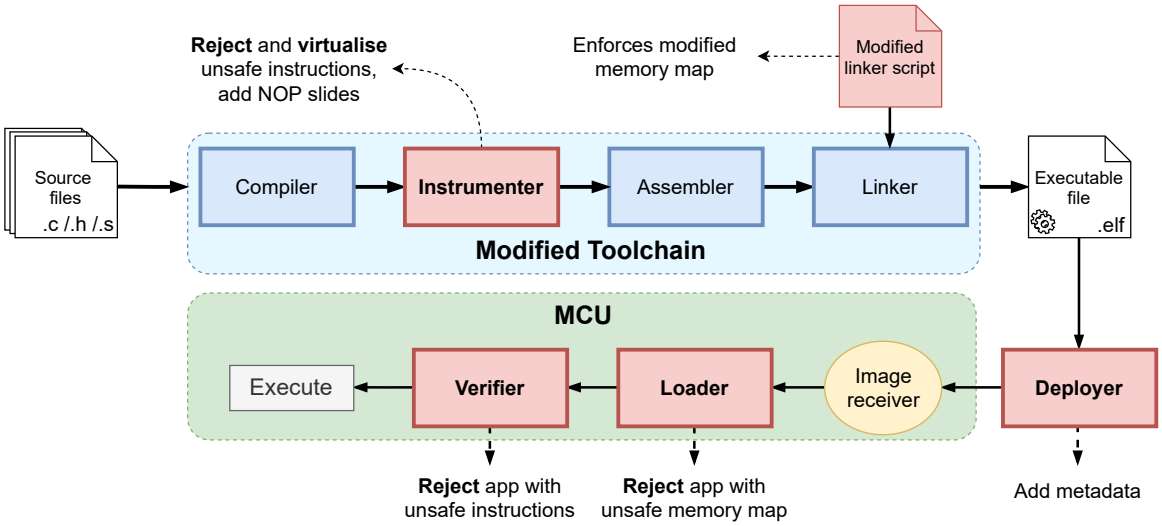


Figure 3.3: PISTIS-enabled Compiler Toolchain.

use the PISTIS-enabled toolchain with the same ease of use as any other toolchain. No extra action is needed as all required steps are performed transparently. Furthermore, developers can simply allow their software to interact with the TCM by invoking any of its valid entry points.

Virtual instructions. Virtual instructions are part of the TCM. They represent a safe replacement to some dynamic unsafe instructions, maintaining the same functionality and adhering to the aforementioned AP. This allows the TCM to verify the safety of the intended operation at run-time before executing the corresponding instruction.

Listing 3.1 shows an example of an unsafe dynamic CALL instruction that tries to jump to an address held by a register. Such an address is only known at run-time. When compiling with the PISTIS-enabled toolchain, such an instruction is replaced by a sequence of instructions shown in Listing 3.2. The main purpose of these instructions is to safely invoke an equivalent safe routine inside the TCM. This routine will check the validity of the target address before jumping to it. The instructions of such a routine are shown in Listing 3.3.

```
...
CALL R10     ; Dynamic call to an address in a register
```

Listing 3.1: An example of a dynamic unsafe CALL instruction.

```
...
DINT              ; Disable interrupts to ensure atomicity
MOV R10, R6       ; copy target address to R6
CALL #safe_call ; Call to a TCM's safe virt. routine
...
```

Listing 3.2: A safe equivalent virtualisation to the CALL instruction in Listing 1.

```
    ...
safe_call:
    CMP #topInstrMem, R6 ; Check upper boundary
    JHS .stopExecution   ; MCU reset if AP is violated
    CMP #btmInstrMem, R6 ; Check bottom boundary
    JL .stopExecution ; MCU reset if AP is violated
    EINT     ; Enable interrupts after passing all checks.
    BR R6 ; Jump to original destination
```

Listing 3.3: An example of a safe virtual function. `safe_call` checks the validity of the original destination before jumping.

Memory isolation: supported operations

Direct Memory Access (DMA). DMA is a crucial feature that enables direct access to the memory without CPU intervention, yielding faster and efficient data transfer between the main memory and other external modules or I/O peripherals. However, DMA is rarely supported by TEEs (e.g. [146, 15, 112]) since it allows bypassing the CPU-controlled access control mechanisms. Notable is that DMA operations in simple embedded systems are always preceded by a CPU-controlled initialisation phase that entails writing some metadata, e.g. source and destination addresses, to some specific MMIO registers. Upon confirmation from the CPU, the DMA controller can execute independently to complete the entire operation. PISTIS is designed to support safe DMA operations by supervising the initialisation phase. First, the instrumenter plugin in the PISTIS-enabled toolchain is configured to instrument the write instructions to the DMA-dedicated MMIO registers. This allows the native DMA operations to be transparently converted into safe hyper-calls to the TCM. Second, the Loader/Verifier module in the TCM is extended to check for unsafe accesses to the DMA registers, i.e. without the use of our safe hyper-calls. Consequently, any untrusted application that tries to bypass our instrumentation is caught before being executed. Finally, at run-time the TCM prevents any direct access to the DMA registers with the help of the aforementioned virtual functions. Thus, safe DMA transfers can be guaranteed in

a similar way to the control-transfer instructions, enabling execution if both the source and destination addresses are valid w.r.t. the aforementioned AP.

Interrupts. Interrupts enable the preemption of the normal application flow to execute user-defined functions, called Interrupt Service Routines (ISRs). These routines are fetched by the CPU depending on the Interrupt Vector Table (IVT), a priority-ordered list containing entry points to all defined ISRs. Supporting interrupt operations is important for many safety-critical application scenarios. Therefore, PISTIS supports preemptive execution of TCM's software modules except for virtual functions (as shown in Listing 3.3) as they are very small (each contains less than 10 instructions) and it is important for the security to execute them atomically.

To enable safe interruption, PISTIS supports a secure context switching mechanism through a chain of function-calls. To do so, PISTIS considers duplicating the IVT and moving both versions into its core memory (the TCM memory). All entry points inside the original IVT are overwritten to point to a unified secure ISR that is maintained inside the TCM memory as well. When invoked, this ISR backs up the MCU state to a fixed safe memory location. It also clears any sensitive data that is accessible by the untrusted software, e.g. contents inside registers. It then triggers an IVT lookup in the other non-modified version of the IVT to execute the target ISR by the interrupt operation. In principle, any ISR represents a set of instructions that is included as a part of the deployed binary image. Thus it can be instrumented (during compilation) and verified by the TCM at load-time. PISTIS instrumentation also adds a call to a virtual function (stored in the TCM memory) at the end of each ISR. When executed, this function safely restores the CPU state. Please note that the modified linker script in the PISTIS-enabled toolchain holds the new address of the modified IVT to maintain the intended functionality of compiled applications.

Memory isolation: variable length instructions

The Von Neumann architecture supports a variable-length instruction set, i.e. some instructions can be of variable lengths, holding one or more words. $\mathcal{A}$dv-s can maliciously leverage this feature to arbitrarily execute code and bypass the memory protection enforced by PISTIS. In other words, some benign-like instructions can behave maliciously by jumping into the middle of a multi-word instruction, located in the insecure memory zone. The target location can hold a value that could be executed as an instruction, allowing for illegal access to the protected memory area. Therefore, it is crucial for

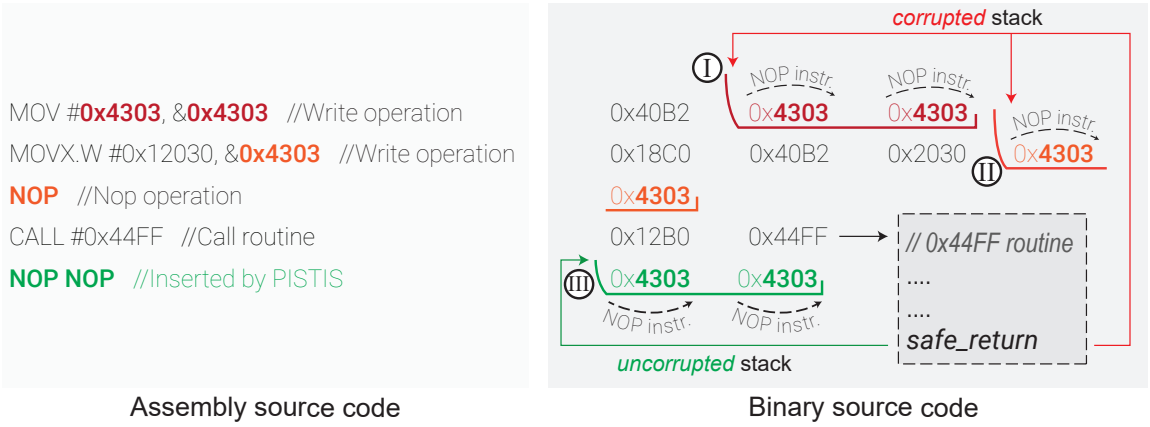


Figure 3.4: Binary code (right) of the assembly source code (left) has 3 NOP slides: a legitimate slide (III) after a `CALL` statement and two *accidental* slides (I) and (II) in the middle of two instructions. (III) is inserted by PISTIS. However, (I) and (II) are either maliciously inserted or accidentally derived from combinations of opcodes and operands. The `safe_return` virtual function only allows returning to a location with a slide. In case of a corrupted stack, the control flow can at most be diverted to another location containing an accidental slide. However, the instruction executed after the slide will always be safe since verified.

PISTIS to only allow jumping to verified instructions. While the TCM can easily check static jumps at load-time, a similar run-time check of dynamic jumps incurs a high run-time overhead.

To tackle this issue, we extend PISTIS's memory isolation design in two directions. First, we extend the functionality of the instrumenter plugin (in the PISTIS-enabled toolchain) to insert a special instruction, called *instruction canary*, before any valid address that can be a target for a potential jump. This instruction will be in the form of a NOP slide: a sequence of two NOP instructions. Given that our modification to the toolchain limits the number of dynamic jump instructions when possible, only a few NOP slides are inserted in each application. Second, the TCM is configured to take such NOP slides into account when virtualising jump instructions.When a dynamic jump executes at run-time, the corresponding virtual safe routine inside the TCM will check whether a NOP slide precedes the target address. If so, the jump is valid (i.e. the jump target is a permitted address inside the insecure memory area). Otherwise, an MCU reset is performed as a consequence of an invalid target address.

Considering an implementation for the MSP430 architecture, Figure 3.4 shows a snippet code, highlighting the working mechanism of the two-NOP canary. Please note that PISTIS does not enforce control flow integrity. Nevertheless, it guarantees that

diverting control flow does not break the maintained memory isolation property.

Security services

Leveraging memory isolation, PISTIS features two important security services, namely $\mathcal{RA}$ and secure code update. Notable is that the design of other security services, e.g. secure erasure, can be easily supported in PISTIS.

Remote Attestation ($\mathcal{RA}$). Considering that the TCM memory is neither writable nor readable, we leverage it to design an $\mathcal{RA}$ service. $\mathcal{RA}$ is a security service that allows a remote verifier to verify the integrity of a prover, such as a piece of software. This technique is commonly used to check whether an embedded device is running the expected software or has been tampered with. To do so, a secret key (that is pre-shared with the verifier) and an integrity verification function (based on MACs) are installed in the TCM memory. The first instruction of this function is considered a valid entry point to PISTIS. Whenever an attestation request is received, this function computes a digest (MAC value) of the entire memory and then sends it back to the verifier for verification. Either a nonce or a time-stamp can be used to avoid replay attacks. Our $\mathcal{RA}$ is similar to SIMPLE [16]. However, it has the advantage of being interruptible as explained in Section 3.4.2.

Although $\mathcal{RA}$ is commonly used in embedded systems, its necessity may seem redundant in the presence of PISTIS, which enforces immutability on untrusted code to ensure compliance with the AP depicted in Figure 3.2. However, an untrusted application can still request a code update, which, despite verification by the TCM, may be malicious. Additionally, $\mathcal{RA}$ can detect rollback attacks and be configured to verify specific memory regions, offering a powerful fine-grained tool for the remote verifier.

Secure code update. We also extended the TCM to include a secure code update service that complies with SUIT [174], an IETF standard for software updates on IoT devices that requires maintaining authenticity, integrity, and confidentiality of the deployed software. By extending the cryptographic module of the TCM with a decryption function, our secure code update mechanism meets the above requirements. Considering a pre-shared secret key, the verifier has to send the software image encrypted along with a MAC value. The image will be deployed if it is decrypted and verified (in terms of matching MAC values and checking the compliance w.r.t. the AP of PISTIS) successfully.

3.5 Formalisation

Formally verifying both the design correctness and implementation are important to provide strong security guarantees w.r.t. the aforementioned threat model (Section 3.4.1). Given that verifying the implementation is platform-dependent and requires continuous efforts with each software update to the PISTIS code, we leave it as future work. We rather focus on verifying the correctness of the design as it is platform-independent and gives more trustworthiness to the proposed solution.

In order to formalise the PISTIS design and memory map layout described in Figure 3.2, and prove it preserves the memory isolation, we need first to introduce some basic concepts. Let us assume for simplicity that we have a memory M that contains the code and where data can also be stored, and that there are a finite number of registers for the program counter `PC`, the stack pointer `SP`, and for intermediate results (e.g. $\texttt{R}_\texttt{j}$ for some `j`). Moreover, let us also assume there is an Interrupt Vector Table (IVT) that contains the addresses of the first instruction of the Interrupt Service Routines (ISRs).

Any application can be thought as composed of sequences and combinations of the following *primitive instructions*:

Definition 1 (Primitive (unsafe) instructions)**.** The set of *primitive (unsafe) instructions* is the following (we assume the address i is a valid address for the memory M):

- `Read(`M`,`i`)` that reads the content of memory M at address i: it denotes $M[i]$;
- `Write(`M`,`i`,`V`)` that writes V in memory M at address i: it denotes $M[i] = V$;
- `Goto(`M`,`i`)` that modifies the program counter `PC` to contain the address i of the given memory M: it denotes `PC =` i;
- `End` that terminates the execution of the entire application;
- Primitive operations (e.g. `and`, `add`, `comparison`) operating on registers/constants, and assignment (`=`) to a register.

An *application* P is a sequence and combination of the above primitive instructions, together with the `IVT` table specifying the addresses of the interrupt service routines.

We can safely assume that all the instructions of a typical MCU can be written in terms of these basic primitives. Let us consider for instance some instructions taken from the MSP430 MCU's family:

- `CALL funcname`: the execution of this instruction stores in the stack (in the memory) the current `PC`, modifying the `SP` to create space for storing the `PC`, and then modifies the `PC` to point to the address corresponding to `funcname` in the memory. Thus it

corresponds to `SP = SP-1`, to create space in the stack, followed by `Write(M,SP,PC)`, to write the `PC` in the stack, followed by `Goto(M,funcname)` to update the `PC` and continue the execution from address `funcname`.

- `RET`: corresponds to `R = Read(M,SP)` followed by `SP = SP+1`, and finally `Goto(M,R)`, using the value stored in register `R`.

The primitive instructions also allow the modelling of more complex instructions like `PUSH` and `POP` in a very similar fashion, as well as different addressing modes (e.g. indexed, symbolic, indirect, absolute as for instance supported by the MSP430 MCU's family). For instance,

- `MOV 0x22(R3), 0x10(R4)` corresponds to `Write(M,0x10 + R4, Read(M, 0x22 + R3))`;
- `ADD 0x22(R3), 0x10(R4)` corresponds to `Write(M, 0x10 + R4, Read(M, 0x22 + R3) + Read(M, 0x10 + R4))`;
- `MOV @R4, &0x3000` corresponds to `Write(M, 0x3000, Read(M, R4))`;
- `MOV 0x22, &0x3000` corresponds to `Write(M, 0x3000, Read(M, PC + 0x22))`.

Given the above basic concepts, to proceed in the formalisation of the PISTIS design, the memory map layout described in Figure 3.2 (right), and the access policy AP (for reading, writing, and jumping), we need to:

i) refine the memory M distinguishing between the persistent memory (M_P), the volatile memory (M_V), and the memory mapped IO (M_{MIO})[6];

ii) explicitly introduce the finite set $EnPo = \{$ `i : word[N]` $\}$ of addresses for the PISTIS core instruction area as specified by the access policy;

iii) and finally introduce the V_{IVT}: `array [K] of word[N]` - the interrupt vector `IVT` of size K.

Moreover, we also need:

- utc_b, utc_e - start and end addresses of the PISTIS core;
- aim_b, aim_e - start and end addresses of the App Instruction Memory;
- $arom_b$, $arom_e$ - start and end addresses of the App Read-Only Memory;
- $utdm_b$, $utdm_e$ - start and end addresses of the PISTIS Data Memory;
- adm_b, adm_e - start and end addresses of the App Data memory;
- $mmio_b$, $mmio_e$ - start and end addresses of the MMIO Registers.

[6]Without loss of generality, we consider each memory as an array of size $2^{\mathtt{N}}$ of bit vectors of size `N` (i.e. `array word[N] of word[N]`), although only a small part might be used. The addresses are bit vectors of size `N` (i.e. `word[N]`. We use 0_N to represent the unsigned word of size N corresponding to value 0, similarly 2_N^{N-1} to represent the unsigned word of size N corresponding to value 2^{N-1}.

The following constraint formalises the memory layout and the non-overlapping of the different memory areas.

$$
\begin{aligned}
&0_N < utc_b < utc_e < aim_b < aim_e < arom_b < arom_e \leq 2_N^{N-1} \\
&0_N \leq utdm_b < utdm_e < adm_b < adm_e \leq 2_N^{N-1} \\
&0_N < mmio_b < mmio_e \leq 2_N^{N-1}
\end{aligned}
$$

To model the constraint that the execution of PISTIS core instructions only happens through pre-defined Entry Points, we need a predicate $EP(i)$ which is true iff the address i is $utc_b \leq i \leq utc_e$ and i is an Entry Point address $EP_j \in EnPo$ for the PISTIS core memory.

$$EP(i) \leftrightarrow ((utc_b \leq i \leq utc_e) \wedge (\bigvee_{EP_j \in EnPo} i = EP_j))$$

Then, to formalise the read/write/jump policies as those enforced in the Figure 3.2 (right), we define the following terms:

$$
AP_R(i, M) \leftrightarrow \begin{pmatrix} (M = M_P \rightarrow (arom_b \leq i \leq arom_e)) & \wedge \\ (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \end{pmatrix}
$$

$$
AP_W(i, M) \leftrightarrow \begin{pmatrix} (M = M_V \rightarrow (adm_b \leq i \leq adm_e)) & \wedge \\ (M = M_{MIO} \rightarrow (mmio_b \leq i \leq mmio_e)) & \end{pmatrix}
$$

$$
AP_X(i, M) \leftrightarrow \Big((M = M_P) \wedge (EP(i) \vee (aim_b \leq i \leq aim_e))\Big)
$$

Moreover, we need also to ensure that each element of the `IVT` table is a valid address w.r.t. the access policy, so that:

$$AP_{IVT}(M) \leftrightarrow \forall i. AP_X(V_{IVT}[i], M)$$

We denote with AP the access policy resulting from the memory layout, from the read/write/jump constraints, and from the fact that each $i \in EnPo$ is such that $utc_b \leq i \leq utc_e$. Given the above formalisations, we can formally define when an application preserves memory isolation w.r.t. a given access policy AP.

Definition 2. Given a memory layout and an access policy AP, an application P *preserves memory isolation w.r.t. the access policiy* AP iff any of its read/write/jump instructions in all possible executions is such that the addresses used in such instructions satisfy the given access policy AP.

We remark that the primitive instructions in Def. 1 do not enforce particular restrictions on the addresses where to read/write or jump (it suffices them being valid addresses). As thoroughly discussed, if the addresses of the application are constant, then checking whether the application preserves memory isolation is trivial. Indeed, it suffices to check whether each address in the application satisfies AP. However, as noted, in many cases such addresses are results of the execution of the application itself, thus the check can only be performed during the execution of the application.

To enforce memory isolation, for a given access policy AP, we can define safe variants of the read/write/jump instructions that will guarantee at run-time that no violation of the access policy occurs.

Definition 3 (Safe primitive instructions)**.** Given an access policy AP, the *safe* read-/write/jump *primitive instructions w.r.t. AP* are:

- $\texttt{Read}_{\texttt{sf}}(M,i)$ that reads the content of memory M at address i if address i is such that read policy $AP_R(i, M)$ holds, otherwise it ends execution;
- $\texttt{Write}_{\texttt{sf}}(M,i,V)$ that writes V in memory M at address i if address i is such that write policy $AP_W(i, M)$ holds, otherwise it ends execution;
- $\texttt{Goto}_{\texttt{sf}}(M,i)$ that modifies the program counter PC to contain the address i if address i is such that branch access policy $AP_X(i, M)$ holds, otherwise it ends execution.

The following theorem holds for the *safe primitive instructions* defined as above.

Theorem 1. Given an access policy AP, the *safe primitive instructions* $\texttt{Read}_{\texttt{sf}}$, $\texttt{Write}_{\texttt{sf}}$, and $\texttt{Goto}_{\texttt{sf}}$ w.r.t. such AP preserve memory isolation and do not allow to violate AP.

The proof directly follows from the definition of the *safe primitive instructions* w.r.t. an AP (see Appendix A.3.2 for the proof). If AP is violated, then each instruction results in ending the execution of the entire application, thus preventing access to memory areas forbidden by AP. On the other hand, if the address satisfies AP, instruction specific access to the specified memory location is allowed.

Given this, we can prove that any application P such that

i) $AP_{IVT}(M)$ holds (i.e. each address $i \in V_{IVT}$ satisfies $AP_X(M, i)$), and

ii) uses only the safe primitive instructions,

preserves memory isolation.

Theorem 2. Let AP be an access policy, P be an application specified with the set of (unsafe) primitive instructions. Let P_{sf} be the application obtained from P by replacing

each of the unsafe primitive instructions with the corresponding safe primitive instruction. If $AP_{IVT}(M)$ holds, then P_{sf} preserves memory isolation, preventing accessing memory addresses or executing code that violates AP.

The proof is by induction on the structure of the application P_{sf} leveraging on Theorem 1 (see Appendix A.3.1 for the proof). We remark that enforcing each element of the `IVT` to satisfy AP ensures that also the interrupt routines are located in memory locations allowed by AP. This requirement can be relaxed by not only rewriting the unsafe read/write/jump instructions with the corresponding safe ones, but also adding for each interrupt routine a new wrapping interrupt routine and modifying the `IVT` to point to the respective wrapping interrupt routine. Each wrapping interrupt routine first checks that the target address is a safe one, and if so, it does the jump to the original address in the non-modified `IVT` copy. Otherwise, it ends the execution.

3.6 Implementation

A prototype of PISTIS is implemented for the MSP430 architecture from Texas Instrument [71]. MSP430 MCUs are based on a Von Neumann memory model, and they are widely used in many critical application domains. For instance, they are employed in Implantable Medical Devices (IMDs), e.g. pacemakers, that have support for standard interfaces for wireless communication [224, 121, 70]. They also have been a target for many research prototypes, including SANCUS [183], SMART [83], and VRASED [185].

In particular, we implemented PISTIS on top of the MSP430F5529 MCU, which features ∼132 kB of FLASH, ∼8 kB of SRAM, and up to an 8 MHz of CPU speed using internal oscillators. We used HMAC-SHA256 and ChaCha20 from the HACL library [273] to implement $\mathcal{RA}$ and secure code update services respectively.

3.6.1 PISTIS in numbers

Our PISTIS implementation is two-fold: we provide a software-based TCM for MSP430F5529 MCU and a GCC compiler plugin that produces PISTIS-compliant binary images. The TCM is modular in the sense that it is composed of a core and various TAs that can be deployed independently by adjusting some configurations in a custom Makefile which serves as an API. The TCM core, including a boot code, Loader/Verifier, virtual functions, the interrupt context switch, and some helper functions, is composed of 810 lines of C code along with 703 lines of Assembly. The cryptographic primitives used, namely HACL HMAC-SHA2 and ChaCha2, comprise 944 lines of code, extracted from the

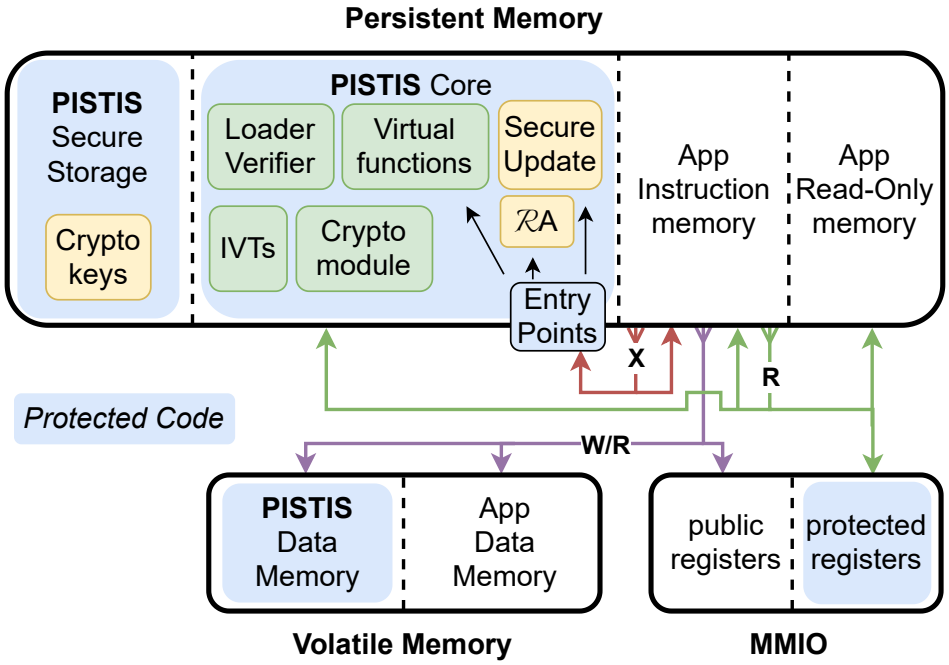


Figure 3.5: PISTIS memory map on MSP430 architecture.

HACL library [273]. Leveraging these primitives, $\mathcal{R}$A implementation includes 78 lines of code, whereas the secure code update mechanism is composed of 220 lines of C code. The MSP430 architecture features 60 main instructions, of which, we had to instrument 24 instructions to maintain strong isolation guarantees (further details can be found in Appendix A.1). We provide an API, comprising a 102-line Makefile in addition to an extended MSP430F5529 linker script with 257 lines, that fully automates the deployment of the TCM on the target device. Furthermore, our GCC plugin serves as another API to fully automate the production of PISTIS-compliant binary images. It comprises a 71-line Makefile that transparently modifies 15 lines of the original linker script of applications and executes 745 lines of python scripts to re-write instructions and embed meta-data. Furthermore, the user API includes a 236-line python script that automates the deployment of produced binaries using serial communication.

3.6.2 Memory map in practice

Section 3.4 elaborated on the design rationale of enforcing memory isolation between software modules. As a consequence, Figure 3.2 visualised the resulted memory map w.r.t. the employed access policy (AP). Given that the AP is mainly concerned with read, write, and execute rights, the number of application instructions requiring virtualisation might vary depending on the application nature. To optimise PISTIS for our

target MSP430 architecture, and thus reduce the number of virtualised instructions, we follow a slightly different memory map that, nevertheless, has equivalent security guarantees to the one shown in Figure 3.2. The new memory map, visualised in Figure 3.5, introduces a secure storage segment, where all the sensitive data, including cryptographic keys, are stored. This memory part is only accessible by the PISTIS core, relaxing the application access policy as follows: (i) Read access is extended to cover the PISTIS Core, App Instruction memory, and PISTIS Data memory; and (ii) Write access is extended to cover PISTIS Data memory.

While our general design solely relies on the virtualisation and verification of the application code to ensure compliance with the AP of PISTIS, our implementation leverages two existing commodity hardware features in MSP430 MCUs: the Bootloader Section (BSL) and the Flash memory controller. Further details follow.

Secure storage using BSL

The BSL is a small memory segment, located as a part of the Flash memory, whose confidentiality and integrity are hardware-enforced by the MCU. It triggers a reset at any illegal access. A legal access occurs through a few entry points, i.e. the Z-area, which can be configured during the physical deployment of PISTIS. Leveraging its intrinsic properties, we customise part of this segment to form the PISTIS secure storage, thus blocking any access by the untrusted application. The Loader/Verifier module in PISTIS is only required to check that the application instructions do not jump to the Z-area. Only the TCM can freely access the Z-area. This allows PISTIS to safely enable read access to the rest of the Flash memory without breaking the security.

Integrity of PISTIS core

While the untrusted software is allowed to read any part of PISTIS core memory, it must not be allowed to have write access to it in order to preserve the integrity of the TCM. One possibility is to follow the guidelines of our design and virtualise all write instructions. However, the MSP430 architecture offers a hardware feature that allows achieving the same goal with better performance. The Flash memory controller regulates all write accesses over the Flash. It must be set up via custom memory-mapped registers that enable (unlock) or disable (lock) writing to the Flash according to their loaded value. In particular, writing to the Flash will only have an effect if there is an instruction that unlocks the Flash controller, i.e. by writing a specific pre-defined byte sequence (password) into one of the controller registers. Thus, PISTIS prevents

the untrusted application from writing to any part of the Flash memory by checking that none of its instructions try to unlock the Flash memory controller. This reduces the number of instructions that have to be virtualised and fully preserves the integrity of PISTIS core. Please note that the volatile Data memory (SRAM) is not controlled by the Flash memory controller. Therefore, to reduce the performance overhead, PISTIS allows applications to read from or write to any part of it. PISTIS rather prevents the leakage of any sensitive data by clearing all shared memory areas before any context switch from a privileged to a non-privileged mode.

3.7 Experimental evaluation

The variety of embedded applications along with the lack of suitable benchmarks make the evaluation of a generic Trusted Computing architecture such as PISTIS non-trivial, requiring a careful selection of a representative set of applications that would reflect the overhead incurred by PISTIS from various perspectives.

We chose a set of 13 applications, considering two main factors: (i) the compatibility with our target experimental platform (MSP430F5529LP), and (ii) making a good balance between the more CPU-intensive and the more IO-intensive tasks that are typical for an embedded device. The source and main functionalities of each application are described in Appendix A.2. Our evaluation metrics include: memory footprint, execution time, deployment time, and power consumption.

NOTE. Unless otherwise specified, applications are compiled using −O3 optimisation flag and 8 MHz as a CPU speed.

3.7.1 Memory footprint

Increased size of instrumented applications. Needless to say that instrumentation adds extra bytes to the size of each application due to inserting extra instructions, e.g. NOP slides, virtual calls, etc.

Considering a GCC toolchain with a −O3 optimisation flag (that further optimises the execution time), Table 3.2 shows the size differences (in bytes) between binaries compiled using a standard GCC toolchain (Orig.), and a PISTIS-enabled GCC toolchain (Mod.) from two perspectives:

- The first two columns compare the sizes of the produced Executable and Linkable Format (ELF) files. ELF is a common standard file format for executable files, object code, and shared libraries, which tells the bootloader how to parse the

Table 3.2: ELF binary size and memory footprint comparison between original applications (Orig.) and PISTIS-enabled ones (Mod.).

App	ELF Binary		Memory Footprint	
	Orig.	Mod.	Orig.	Mod.
SerialMSP	3884 B	412 B (**-89.39%**)	302 B	356 B (+**17.88%**)
CopyDMA	5764 B	694 B (**-87.96%**)	444 B	628 B (+**41.44%**)
XorCypher	5940 B	532 B (**-91.04%**)	247 B	475 B (+**92.31%**)
Bitcount	5664 B	1602 B (**-71.72%**)	3684 B	5462 B (+**48.26%**)
SHA-256	9448 B	5518 B (**-41.60%**)	1376 B	1546 B (+**12.35%**)
ML-acc	16616 B	9512 B (**-42.75%**)	6174 B	9452 B (+**53.09%**)
PrimeFactor	33200 B	3650 B (**-89.01%**)	2192 B	3286 B (+**49.91%**)
32bitMath	6036 B	822 B (**-86.38%**)	522 B	766 B (+**46.74%**)
16bitSwitch	3940 B	182 B (**-95.38%**)	102 B	126 B (+**23.53%**)
8bitMatrix	4640 B	916 B (**-80.26%**)	844 B	860 B (+**1.90%**)
MatrixMul	4324 B	572 B (**-86.77%**)	500 B	516 B (+**3.20%**)
firFilter	24912 B	5486 B (**-77.98%**)	3312 B	5430 B (+**63.95%**)
dhrystone	7840 B	2468 B (**-68.52%**)	1335 B	2411 B (+**80.60%**)
Average		**-77.60%**		+**41.17%**

received image, extract the real binary, and deploy it. Therefore, the size of ELF binary images is larger than native binaries since they include many extra meta-data. In the case of PISTIS, our toolchain customizes the relevant included meta-data, resulting in the reduction of the size of ELF binaries by an average of 77.6% as shown in Table 3.2. Please note that in embedded systems, applications are compiled statically as the entire binary image is considered self-contained with no need for dynamically-linked libraries. Therefore, all the libraries used in any application can be compiled using the PISTIS's compiler framework.

- The other two columns in Table 3.2 compare the real binary sizes of both standard and instrumented binary images when deployed. In other words, the recorded sizes represent the amount of consumed Flash memory in bytes. The increase of binary sizes when considering PISTIS averages at 41.17%.

PISTIS memory footprint. As shown in Figure 3.5, PISTIS persistently occupies part of the Flash memory. The size of the entire PISTIS software is 19.5 kB (of which 11 kB for the TCM core), amounting to 14.7% of the available Flash memory (∼132 kB) on the target MCU.

3.7.2 Deployment time

In PISTIS, deploying any application occurs in two phases. First, the binary image is transferred as a stream of packets, starting with some meta-data, i.e. size, Cyclic Redundancy Check (CRC) value, etc. Second, the TCM's Loader/verifier module is executed to verify both the integrity of the binary image w.r.t. the received CRC value and the safety of its instructions w.r.t. the AP of PISTIS. If both checks are passed, the control is then given to the first instruction of this application, indicating a successful deployment.

Considering a serial UART communication medium with a speed of 9600 bps, Table 3.3 compares the deployment overhead incurred in each phase when deploying applications using either a standard compilation framework or the PISTIS one. Due to the customised ELF binary format used by PISTIS that further optimises the size, the transfer time is faster, reducing the overhead by an average of no less than 75.4%. On the contrary, PISTIS increases the verification time by an average of around 37.56% due to the load-time verification mechanism employed. Nevertheless, the total deployment time when considering PISTIS is less than the time consumed in normal deployment procedures, averaging at -73.63%.

Secure deployment overhead

So far, the description above did not consider the properties of the secure code update mechanism recommended by SUIT [174]. It only took into account the validation of a CRC32 value. Our secure deployment mechanism entails encrypting the target binary image at the transmitter side and decrypting it at the receiver side. This is in addition to including a MAC value that reflects the integrity status of the image during transmission. Comparing to the non-secure deployment mechanism in PISTIS, the secure code update slightly increases the transfer-time overhead due to including some extra bytes as a MAC value. In particular, 28 bytes are added to each binary image as a result of replacing CRC32 with an HMAC-SHA256 cryptographic primitive, increasing the overhead from -77.52% to -77.09%. Furthermore, the decryption and MAC verification increase the verification time overhead from 37.56% to78.82%, resulting in an increase of the overall deployment time overhead from -73.63% to -71.71%. Nevertheless, the secure code update mechanism of PISTIS still outperforms the normal deployment procedure.

Table 3.3: Deployment (code update) time overhead when considering PISTIS (Mod.) compared to normal procedures (Orig.).

App	Transfer		Verification		Total
	Orig.	Mod.	Orig.	Mod.	
SerialMSP	3242 ms	347 ms	51 ms	62 ms	**-87.58 %**
CopyDMA	4809 ms	585 ms	92 ms	111 ms	**-85.80 %**
XorCypher	4957 ms	448 ms	42 ms	69 ms	**-89.66 %**
Bitcount	4727 ms	1340 ms	646 ms	913 ms	**-58.07 %**
SHA-256	7879 ms	4605 ms	212 ms	261 ms	**-39.86 %**
ML-acc	13849 ms	7930 ms	1534 ms	2109 ms	**-34.74 %**
PrimeFactor	27670 ms	3049 ms	597 ms	716 ms	**-86.68 %**
32bitMath	5035 ms	689 ms	113 ms	135 ms	**-83.99 %**
16bitSwitch	3289 ms	157 ms	29 ms	34 ms	**-94.24 %**
8bitMatrix	3871 ms	771 ms	172 ms	189 ms	**-76.26 %**
MatrixMul	3608 ms	482 ms	98 ms	196 ms	**-81.71 %**
firFilter	20767 ms	4577 ms	633 ms	910 ms	**-74.36 %**
dhrystone	6537 ms	2059 ms	210 ms	356 ms	**-64.21 %**
Average		**-77.52 %**		**+37.56 %**	**-73.63 %**

3.7.3 Execution time

In order to estimate the run-time overhead that PISTIS imposes on the normal execution of applications, we measured the difference between the time it takes to execute our test applications with and without PISTIS. Measurements are recorded at a CPU speed of 1 MHz, considering an average of 5 different test runs for each one. Table 3.4 shows that PISTIS increases the execution time by an average of 52.72%. The maximum run-time overhead recorded is 127.32%, whereas the minimum one is 0.17%. The high heterogeneity of results is due to the nature of applications. For instance, memory- and CPU-intensive applications, e.g. CopyDMA and ML-acc, require virtualising and verifying many instructions at run-time, incurring high run-time overhead. On the contrary, the I/O-intensive applications, e.g. SerialMSP, contain few unsafe instructions to virtualise, resulting in a slight increase of the execution time. It is important to mention that this is an acceptable price to pay for accommodating a full-featured Trusted Computing architecture without any hardware modification. We also note that using other optimisation flags rather than −O3 might optimise the execution time for some applications. For instance, using −O2 optimisation flag with the PrimeFactor applica-

Table 3.4: Comparison between the execution time of original applications (Orig.) and the execution time of PISTIS-compliant ones (Mod.).

App	Normal Execution (Orig.)	PISTIS-enabled Execution (Mod.)
SerialMSP	334.1976 ms	335.325 ms (+**0.34%**)
CopyDMA	118.4960 ms	238.656 ms (+**101.40%**)
XorCypher	245.6500 ms	446.104 ms (+**81.60%**)
Bitcount	5.7520 ms	5.786 ms (+**0.59%**)
SHA-256	49.1888 ms	89.046 ms (+**81.03%**)
ML-acc	1456.9092 ms	3311.829 ms (+**127.32%**)
PrimeFactor	4.0810 ms	5.938 ms (+**45.50%**)
32bitMath	0.9310 ms	1.294 ms (+**38.99%**)
16bitSwitch	0.0050 ms	0.006 ms (+**20.00%**)
8bitMatrix	0.5760 ms	0.577 ms (+**0.17%**)
MatrixMul	0.3430 ms	0.344 ms (+**0.29%**)
firFilter	1093.5059 ms	2359.619 ms (+**115.78%**)
dhrystone	102.9200 ms	177.336 ms (+**72.30%**)
Average		+**52.72%**

tion reduces the run-time overhead by 9%, compared to the −O3 version. Finally, we note that our $\mathcal{RA}$ consumes 7.4 seconds when computing MAC for a 64 kB memory block using the default clock speed (8 MHz).

3.7.4 Power consumption

IoT devices are often used in areas where a connection to a power line is simply unfeasible or expensive. Therefore, such devices depend on batteries as their main power source, requiring optimised energy usage to reduce maintenance costs.

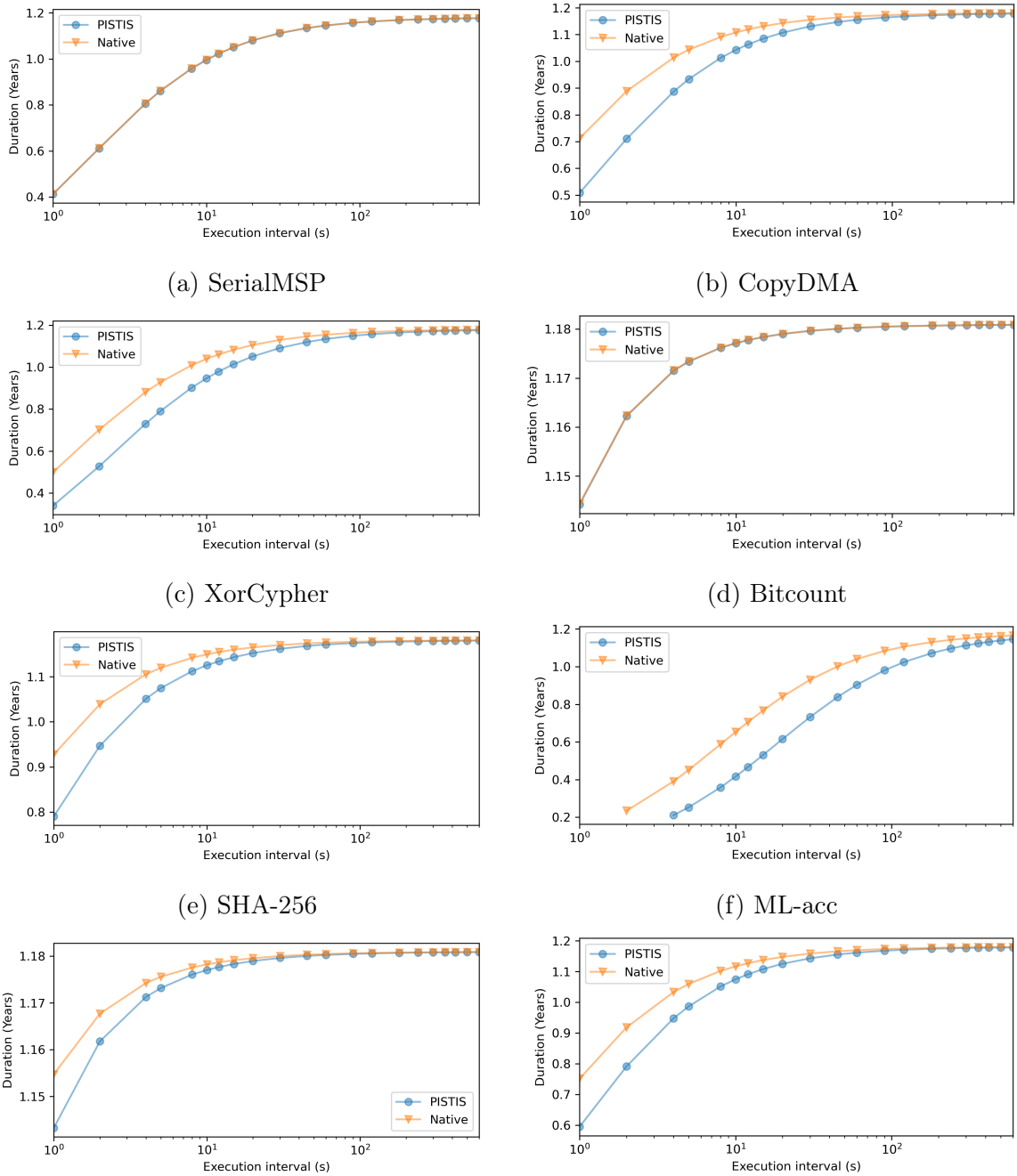
Figure 3.6 compares the energy consumption of applications compiled with and without PISTIS. We evaluate all our demo applications along with TI's Dhrystone benchmark, covering a diverse range of I/O-, memory-, and CPU-intensive operations. Considering a 3.0 V / 3 Ah battery, Figure 3.6 illustrates the impact on on battery lifetime across different execution rates. In the worst-case scenario, where our sample applications run in a continous loop (approximately once every second),battery lifetime decreases by an average of 13.43%. However, when execution is limited to once every two minutes, the impact on battery life drops below 1% across all applications. We also note that the battery would last for no less than 1 year when performing our $\mathcal{RA}$ mechanism once an hour, considering any of the sample applications executing once every 2 minutes.

3.7.5 PISTIS in practice

The reported run-time overhead might seem not acceptable in some scenarios. However, we note that this overhead does not reflect the complete image in real-world scenarios. To do so, we emulated a real case, in which we deployed PISTIS in an MSP430-based device equipped with a gyroscope sensor and a low-power IEEE 802.15.4 compliant radio. The main task of the device was to obtain 10 different measurements from the gyroscope (x,y,z coordinates), encrypt them, transmit the encrypted data over the network to a host controller, and wait for an acknowledgement. By measuring the time required to perform this entire set of actions (that constitutes the job of the IoT device), we noticed that out of a total of 276 ms as a complete execution time, only **6.02%** of the time was occupied by the software that should be instrumented by PISTIS. The majority of the time (93.98%) was consumed by the network layer (that is hosted in another MCU and interfaced with through peripherals). This means that if PISTIS incurs 100% performance overhead on top of the normal execution time of the corresponding software module, the overall performance overhead on top of the total execution time of the entire job will be not more than 9.5%. Therefore, we believe that the run-time overhead incurred by PISTIS is reasonable in practice.

3.8 Summary

This chapter introduced PISTIS, a pure-software Trusted Computing architecture, also known as TEE, for low-end embedded devices. PISTIS provides the core security primitives, such as memory isolation, to enable essential security services such as remote attestation and secure code update. PISTIS provides strong security guarantees while supporting crucial device capabilities, e.g. interrupts and DMA operations. Given that PISTIS is a software-based RoT, we proved the correctness of its memory isolation design formally. This lies the foundation to a solid and comprehensive security stack that, as we will see in the remainder of this thesis, can be leveraged and extended to offer additional security services to various application domains.



(a) SerialMSP

(b) CopyDMA

(c) XorCypher

(d) Bitcount

(e) SHA-256

(f) ML-acc

(g) PrimeFactor

(h) Dhrystone

Figure 3.6: An analysis of the power consumption of eight different exemplar applications.

Chapter 4

MPI: Memory Protection for Intermittent Computing

This chapter is based on a work previously published as:

Grisafi, M., Ammar, M., Yildirim, K. S., Crispo, B. (2022). MPI: Memory Protection for Intermittent Computing. IEEE Transactions on Information Forensics and Security, 17, 3597-3610.

For the sake of conciseness and to avoid excessive redundancy, some parts of the original publication were adapted to better fit the thesis structure. We invite the reader to refer to the original publication for more details.

Preamble

The world of embedded systems encompasses a vast array of use cases and devices. These MCUs are highly heterogeneous and versatile, capable of adapting to multiple environments and applications, ranging from wearables to industrial automation. However, their constrained resources often hinder the deployment of robust security solutions. Interestingly, in some scenarios, the limited hardware of these devices is not the only obstacle to achieving security. One such scenario arises from the absence of a reliable power source, leading to intermittent execution. Batteryless devices harvest energy from sporadic ambient sources, enabling a wide range of long-lived, stand-alone, and environmentally friendly sustainable applications. Software on these devices operates intermittently due to frequent power failures which lead the device to lose its computational state and hinders the forward progress of computation and memory

consistency. One solution to remedy this situation is to pair programs with checkpoints to save a snapshot of the intermediate program state to non-volatile memory before a power loss. Due to the lack of protection mechanisms in state-of-the-art intermittent systems, checkpoints can be altered either by programmer errors or deliberately by an attacker. This situation leads to catastrophic effects since the program execution might be corrupted, and in turn, the device might malfunction.

In this chapter, we introduce MPI (**M**emory **P**rotection for **I**ntermittent **C**omputing), an extension of PISTIS specifically designed for intermittent computing systems. MPI enables a reliable and secure generation and restoration of checkpoints, maintaining their integrity and access control in the presence of remote software-based attacks without trusting the user program or requiring programmer intervention. Notable is that MPI neither requires hardware modifications nor depends on hardware features that might not exist in all batteryless platforms. Our experiments on a real batteryless platform show that MPI provides stronger security guarantees compared to the state-of-the-art approaches, with a comparable time and energy overhead.

Chapter outline. The remainder of this chapter is organised as follows. Section 4.1 introduces the contribution of our work, exploring the issue of secure intermittent computing. Section 4.2 analyses the state-of-the-art on the security techniques employed in this field, while Section 4.3 and 4.4 give an overview of MPI design and implementation, respectively. We provide an extensive evaluation of our technique in 4.5 and summarise our work in Section 4.6.

4.1 Introduction

Progress in energy harvesting circuits and decrease in power requirements of processing, sensing, and communication hardware, free the Internet of Things (IoT) devices from their batteries and enable a broad range of long-lived, stand-alone, and environmentally-friendly sustainable applications. Batteryless IoT devices can operate by harvesting energy from ambient sources such as radio frequency waves [97], vibration [100], and sunlight [166]. Recent works have demonstrated several applications enabled by these devices, such as long-lived wearables [247], robotics [265], body implants [109], inference [96], and even space monitoring [73].

There are several computing platforms (e.g. WISP [214], Flicker [113], and Camaroptera [181, 74]) developed by the researchers. These platorms include an ultra-low-power Microcontroller Unit (MCU) with an embedded non-volatile secondary mem-

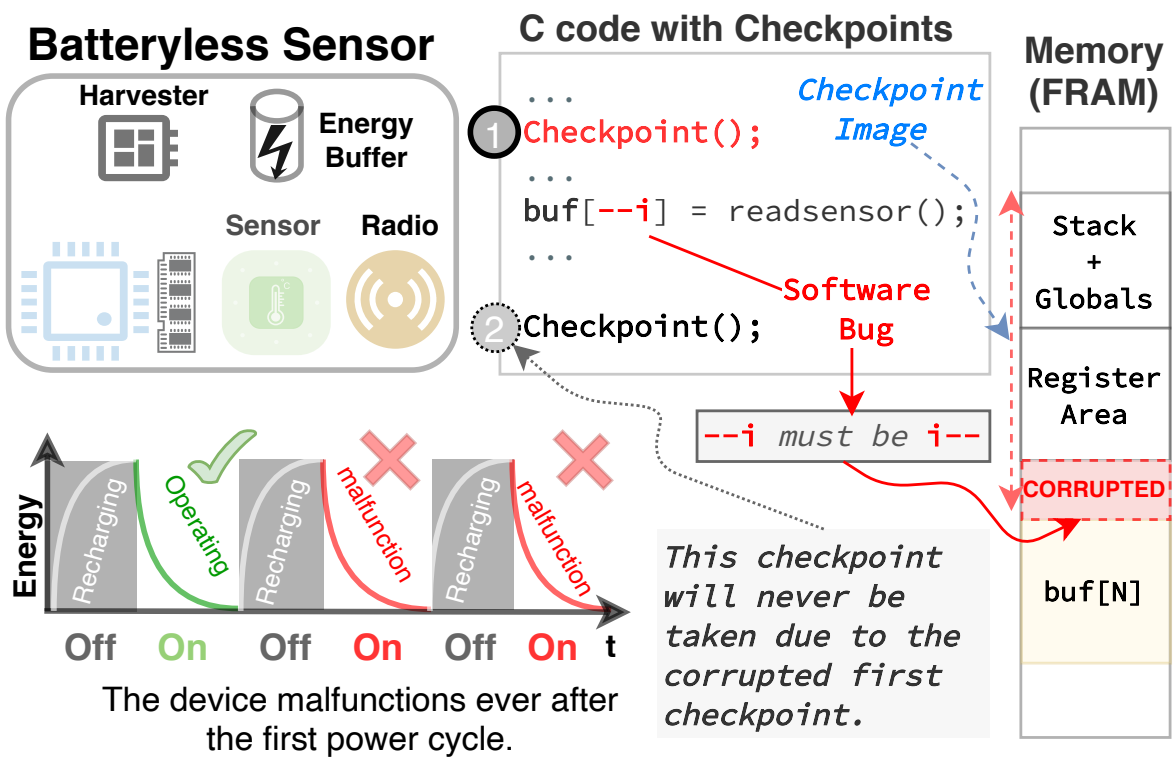


Figure 4.1: A buffer overflow corrupts the first checkpoint. After power failure, the program might continue from an inconsistent memory state that might even lead the device to malfunction and stop operating. Without checkpoints security (to ensure integrity and confidentiality), batteryless devices are prone to catastrophic software errors and attacks.

ory (e.g. FRAM [240]), several sensors, a small capacitor, and an energy harvesting circuit. The capacitor stores the harvested ambient energy and powers all the hardware components. When the stored energy is above an operating threshold, the device reboots to execute a burst of operations, which dissipates the stored energy rapidly. At the depletion of energy, the device turns off due to a power failure that clears the volatile computational state (i.e. the general and special purpose MCU registers and the content of the volatile memory). Therefore, power failures lead to *intermittent execution* of programs, during which the computation is interleaved with intervals when the batteryless device is off and harvesting energy to operate again.

Intermittent execution might hinder the progress of computation due to the loss of volatile computational state [42]. Moreover, rebooting after a power failure and re-executing programs might lead to data, and in turn, memory inconsistencies. One of the approaches to ensure the forward progress of computation and to preserve data and memory consistency is instrumenting programs with either manual or automatic

checkpoints [208, 161, 33, 254, 37, 7, 142]. A checkpoint saves a snapshot of the intermediate program state (i.e. stack, heap, program counter, and volatile registers) to non-volatile memory before a power loss. When the device boots up again, it restores its program state from the most recently captured checkpoint.

Securing checkpoints to protect their confidentiality and integrity is crucial to ensure the correct operation of batteryless devices. Programmer errors might corrupt checkpoints (e.g. erroneous use of pointers might overwrite checkpoints, see Figure 4.1) or an attacker can deliberately alter the checkpoint image. For instance, an attacker with access to an arbitrary memory write primitive, e.g. a buffer overflow in the application, could leverage it to overwrite the checkpoint image. Given that checkpoints must be stored in writable and readable non-volatile memory, they represent a valuable target that can be easily reached by attackers. This situation is catastrophic since the program becomes corrupted, and the device malfunctions ever after. Moreover, checkpoints might include confidential data that can be stolen by an attacker [28]. As of now, only a few studies considered the security issues of checkpointing systems [93, 231, 76]. Unfortunately, these studies pose significant deficiencies we list below.

P1 **Lack of Integrity.** The prior art either does not guarantee checkpoint integrity or guarantees it under an unrealistic threat model that assumes benign software executing on the device. This assumption is weak since malicious or error-prone software (written in unsafe programming languages such as C) can easily alter and corrupt the checkpoints.

P2 **Cryptography Overhead.** Encryption/decryption mechanisms to ensure checkpoint confidentiality (e.g. employed in [93]) introduce excessive time and energy overhead for batteryless devices. It also require additional hardware support (e.g. secure key storage) to protect secret keys to eliminate brute-force and guessing attacks.

P3 **Hardware Requirements.** Recent work [76] eliminates symmetric-key cryptography operations and other additional hardware components by targeting MCUs with Memory Protection Units (MPUs). However, MPUs require error-prone hardware configuration and leave hardware registers or some RAM segments unprotected [112]. Moreover, frequent power failures in batteryless systems might clear the MPU configuration, which disables memory protection and makes the entire memory accessible until it is reconfigured. Another solution in [231] leverages a tamper-free non-volatile memory (i.e. another additional hardware component) to store the checkpoint confidentially without cryptographic operations.

Problem statement. We seek a new security solution for checkpointing batteryless devices that:

- Preserves both the integrity and access control (a.k.a. confidentiality) of checkpoints in the presence of a strong software-based remote adversary that can control the software state of the underlying device;
- Targets the entire spectrum of batteryless devices, i.e. it does not require any specific hardware support (e.g. MPU) and introduces an acceptable energy overhead.

Contributions. This chapter proposes a pure-software intermittent-compliant trusted hypervisor, called **MPI** (**M**emory **P**rotection for **I**ntermittent Computing Systems). MPI provides memory protection and access control mechanisms, guarantees secure generation and restoration of checkpoints, and preserves checkpoints integrity and access control to maintain the forward progress property of the running application. Our target is low-end embedded devices, including the ones that have zero hardware security features. MPI sandboxes its pure-software Trusted Computing Module (TCM) in a virtually isolated and protected memory area. The main task of the TCM is to manage access control over the entire physical memory of the underlying MCU, forcing memory protection in the presence of bugs in other software modules. The TCM memory management's task includes: (i) reliably and securely generating and storing checkpoints when invoked, and (ii) restoring the last valid checkpoint upon power failures. Therefore, the untrusted software can only choose when to generate a checkpoint. It is the responsibility of the TCM to maintain the confidentiality and integrity of the invoked checkpoints. MPI provides memory protection depending on selective machine-level instructions virtualisation and verification. MPI is accompanied with a custom, though untrusted, toolchain for software compilation and virtualisation. The verification of virtualised instructions is either performed at load- or run-time (depending on the type of virtualised instruction) on the device itself, assuring the memory protection guarantees.

In contrast to all existing approaches [93, 231, 76], MPI neither trusts the running (potentially malicious) software nor depends on time- and energy-consuming encryption and decryption mechanisms. It is, therefore, secure and highly optimised for resource-constrained intermittent devices. Furthermore, MPI requires no hardware security features, targeting the entire spectrum of intermittent devices. Additionally, MPI can be extended to a full-fledged hypervisor. For instance, it can offer various lightweight security features such as secure loading of different *untrusted* software modules and remote attestation. We show that MPI provides strong security guarantees, particularly

Table 4.1: A comparison of the main features and characteristics of MPI against relevant studies in intermittent computing.

Related Work (*Intermittent Computing*)	*Task-based or Check-pointing*	*Security*	*Cryptography*	*Hardware Support*	*Manual Configura-tions*	*Checkpoints Access Control*	*Checkpoints Integrity*
Chain [60], Alpaca [162], Mayfly [114], InK [265], Coala [164],	Task-based	No ✗	N/A ✗	N/A ✗	N/A ✗	N/A ✗	N/A ✗
Dewdrop [42], Mementos [208], DINO [161], Hibernus++ [33], Ratchet [254], HarvOS [37], Chinchilla [163], TICS [142]	Checkpointing	No ✗	N/A ✗	N/A ✗	N/A ✗	N/A ✗	N/A ✗
Optimal checkpointing [93]	Checkpointing	Yes ✓	Requires symmetric encryption/de-cryption ✗	Hardware accelerated cryptographic primitives and secure key storage ✗	Yes ✗	Weak ●	No ✗
Secure checkpointing [231]	Checkpointing	Yes ✓	requires computing keyed-based Hash value ✓	Requires tamper-free memory and secure key storage ✗	Yes ✗	Weak ●	Weak ●
SIA [76]	Checkpointing	Yes ✓	No cryptography ✓	Requires Memory Protection Unit (MPU) ✗	Yes ✗	Weak ●	Weak ●
MPI (this work)	**Checkpointi**	**Yes** ✓	**No cryptography** ✓	**Zero hardware** ✓	No ✓	**Strong** ✓	**Strong** ✓

● means that this feature is only guaranteed if the intermittent software module is bug-free.

integrity and access control of checkpoints, without sacrificing the performance in terms of run-time overhead and energy consumption.

In short, this chapter makes the following contributions:

1. **Intermittent Hypervisor:** We design and implement the first pure-software hypervisor that provides lightweight memory protection for batteryless IoT systems.

2. **Checkpoints Integrity and Access Control:** Our hypervisor is the only one that guarantees the integrity and access control of checkpoints simultaneously in batteryless IoT devices, without undermining the adversary.

3. **Open Source Release:** We plan to release MPI as an open-source C library along with a GCC compiler plugin (via [107]) with example code, for the widespread adoption of secure intermittent computing.

4.2 Background and related work

Recent work proposed several batteryless computing platforms. WISP [214] is an early RFID-based computational platform that can perform sensing and computation by harvesting the power transmitted wirelessly by the RFID readers. There are several

different platforrms based on WISP, such as WISPCam [178] that performs battery-free image capture. Flicker [113] is a plug-and-play architecture that supports a wide range of energy harvesting, wireless communication, and sensors. Camaroptera [181, 74] is equipped with an ultra-low-power camera for batteryless remote image sensing. All mentioned platforms include an MSP430FR [239] series ultra-low-power MCU from Texas Instruments, a de facto standard for energy harvesting systems. This MCU series has an address space covering SRAM and FRAM [240] memory components. Compared to Flash memory, FRAM has lower power requirements, faster write performance and greater read/write endurance.

4.2.1 Intermittent computing approaches

Batteryless devices store the harvested energy in their energy storage capacitors to power all the hardware components. Power failures occur when there is no energy in this capacitor, which interrupts the ongoing computation and leads the device to lose the volatile computational state and intermediate results. Upon reboot, the interrupted program starts from the beginning, which hinders the *forward progress* of the computation. Moreover, re-executing code blocks might also lead to *data inconsistency* issues. We classify state-of-the-art approaches that tackle these challenges into *task-based* systems and *checkpointing* systems. In task-based systems [60, 114, 265, 162, 164], the programmers follow a specific programming model to develop their programs. They decompose the computation into a collection of tasks at compile-time, identify the control flow and manage the data shared and manipulated by the tasks. The tasks have atomic all-or-nothing semantics so that their re-execution does not leave non-volatile memory in an inconsistent state. Hence, task-based systems ensure the atomic completion of the tasks despite arbitrary power failures. However, splitting the computation into several tasks and identifying the control-flow and data sharing/communication among the tasks require a significant developer effort, and it is error-prone [142].

Contrary to task-based systems, checkpointing systems are easy to program since they only require instrumenting C programs with either manual or automatic checkpoints [208, 161, 33, 254, 142]. Checkpoints save the current program state in FRAM before power failures. After reboot, the device stores the program state using the latest successful checkpoint data in FRAM. Our focus in this chapter is checkpointing intermittent systems.

4.2.2 Secure intermittent systems

Embedded devices, in general, have a specialised nature in terms of limited resources and computing power. They are employed in many IoT application scenarios, where they process privacy-sensitive data or perform safety-critical operations, making them attractive targets for a wide variety of cyber attacks [144, 255, 246]. Batteryless embedded devices, compared to battery-powered ones, have a larger attack surface due to the inherent characteristics resulting from frequent power failures that make them vulnerable to power transfer attacks and others [212, 128, 161, 206]. Ensuring secure and safe operations on batteryless devices is more challenging due to their limited capabilities and extreme energy budgets. In particular, by altering the checkpoint image, an attacker can manipulate the state of the intermittent program and prevent the device from functioning correctly. Thus, it's crucial for security to protect the confidentiality and integrity of the saved checkpoints.

Current literature proposed three approaches to address the security issues of checkpoints [93, 231, 76]. Unfortunately, none of these approaches provide checkpoint integrity guarantees considering a realistic adversary that can perform software remote-only attacks. In particular, they are not immune to the memory corruption vulnerability shown in Figure 4.1. Furthermore, the work in [93] depends on cryptographic primitives to encrypt and decrypt checkpoints when needed. The frequent use of these operations is time- and energy-consuming, and therefore unsuitable for intermittent devices. Besides, dependency on cryptographic primitives requires either secure key storage (which demands modifying the hardware architectures) or maintaining the secret key in the unprotected non-volatile memory. This situation makes the system vulnerable to brute-force and guessing attacks. A follow-up proposal [231] reduces the use of cryptographic primitives to only computing an integrity-checking value that would verify the freshness and authenticity of the stored checkpoint. It addresses confidentiality by storing sensitive parts of the checkpoint in a tamper-free non-volatile memory, making it relatively more hardware-dependent than the previous one. Unfortunately, both approaches assume benign software running on the underlying device. However, real attacks and vulnerability analyses have demonstrated that this is a weak assumption since attackers keep exploiting memory corruption vulnerabilities in programs written using unsafe programming languages, such as C, by security-unaware developers [63, 149, 144]. Since malware or malicious software can easily compute a correct digest value related to the corrupted and tampered data, proposals [93] and [231] cannot guarantee the confidentiality and integrity properties of checkpoints.

SIA [76] has taken a step further by proposing an architecture for securing intermittent devices. It eliminates energy-consuming cryptographic functions by utilising some basic hardware features in some MCUs to isolate and protect checkpoints. In this regard, SIA targets devices based on MCUs with Memory Protection Units (MPU) or richer hardware features. While this excludes devices that lack any hardware security features, SIA also depends on assumptions, which are hard to meet in practice, to maintain a secure state of intermittent devices. First, SIA stores the checkpoint image along with the program code in a protected memory area by enforcing some access rules using an MPU. However, the program code might contain vulnerabilities that would ease corrupting checkpoints and tamper with their integrity. Second, the secure operation of SIA depends on software developers to configure the various hardware parameters correctly through software with each application deployment, rendering it error-prone. Last, the fluctuating power is a source for a brownout reset that, in some MCUs, clears the entire configurations of MPU registers, disabling protection and making the entire memory accessible [123].

Very recently, an experimental evaluation of different techniques to secure checkpoints in intermittent computing systems has been presented in [28]. The entire evaluation is based on the new generation of Arm Cortex-M MCUs equipped with Arm TrustZone-M [26]. Such MCUs cannot be used in real-world intermittent computing scenarios since they only have Flash memories whose write endurance is limited (i.e. they can be worn off in a few hours as demonstrated in [21]).

Contrary to the prior approaches, MPI is the first that provides the missing memory protection in intermittent embedded devices, which is the main source of data inconsistency and security violations. Table 4.1 provides a summary of the related work in intermittent computing and the alignment of MPI among them.

4.2.3 MPI vs. other memory protection techniques

In the past years, several memory protection mechanisms and Trusted Computing architectures have been proposed to provide memory protection and detection of attacks on embedded devices. Such proposals can be classified into Software-based [15, 146], Hardware-based [184], or hybrid (a.k.a., software-hardware co-design) [84, 140, 185, 112] approaches. Each approach has pros and cons. However, a key problem in all of them is that their current design does not support intermittent computing. In other words, these architectures guarantee their security properties in the power-on state. Nevertheless, the design of the memory isolation technique in the aforementioned software-based

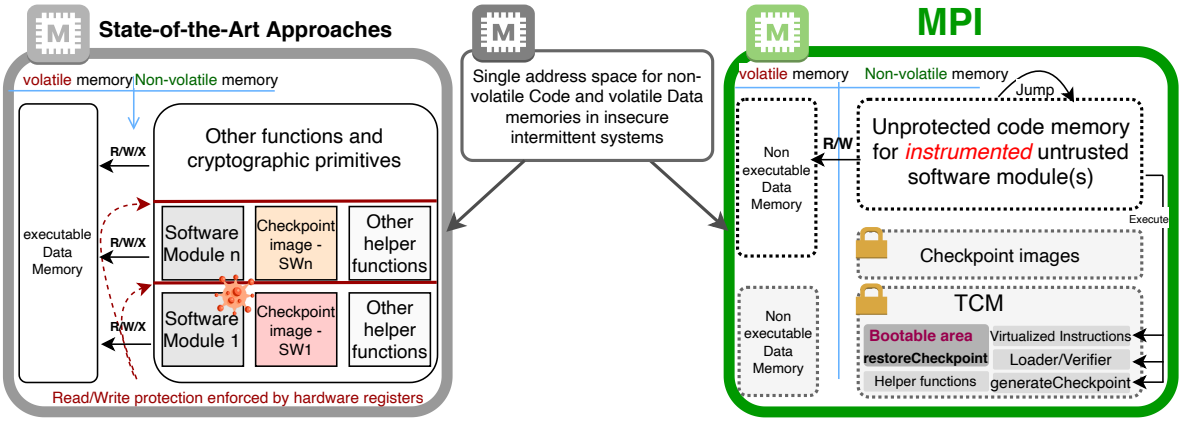


Figure 4.2: An overview of the MCU memory layout when applying (1) the state-of-the-art solutions and (2) MPI to single address space insecure intermittent systems. In MPI, dashed boxes represent the logical isolation enforced. State-of-the-art approaches have several limitations: (i) they are prone to malware infestations and they cannot strongly preserve confidentiality and integrity; (ii) not all low-end embedded devices can support MPU, hardware accelerators for encryption, and key storage; (iii) MPU configuration is error-prone and difficult for security-unaware developers; and (iv) cryptography operations are energy and time consuming operations.

approaches partially overlaps with MPI. For instance, Harbor [146] is a design and implementation of a hypervisor for the SOS Operating System, a dynamic operating system for sensor nodes [111]. It provides coarse-grained memory protection, inspiring from the Software Fault Isolation approach (SFI) [259]. In addition to the lack of support to intermittent computing, Harbor suffers from a high run-time overhead and the need to trust the compiler toolchain. The Security MicroVisor (SμV) [15] has been proposed as a software-based memory isolation mechanism for bare-metal embedded devices. It provided further improvement compared to Harbor in the sense that it does not trust the compiler toolchain and incurs less run-time overhead. Compared to MPI, the SμV does not support intermittent computing. Furthermore, the SμV has targeted the Harvard architecture which is less challenging than the Von Neumann one (the underlying one of MSP430 devices) in terms of having fixed-length instructions and physically separated Data (RAM) and Program (Flash) memories. The lack of these two properties in the targeted architecture by MPI adds more challenges to its design and implementation.

4.3 MPI

MPI is a hypervisor-based approach aimed at providing lightweight memory protection for batteryless platforms. It guarantees integrity and access control for checkpoint images in both power-on and power-off states, regardless of the hardware features of the underlying MCU. Being an extension of PISTIS, MPI consists of two main blocks. The first block is the actual hypervisor software deployed in the target device itself and serves as a trust anchor, the so-called *Trusted Computing Module* (TCM). The second block is a custom compiler toolchain that generates MPI-compliant binary software images. In particular, the custom compiler toolchain will re-write and instrument any software at the instruction level to adhere to the memory protection policy of the TCM. The TCM, which should be the first component deployed in the device, does not trust any toolchain. It does not allow any other software module to be deployed within the same memory address space without double-checking its safety w.r.t. its memory protection policy. To narrow the focus, the TCM logically divides the memory into several zones with different privilege levels, where untrusted software is deployed in a non-privileged memory area. Checkpoint images are stored in a protected memory area that cannot be read or written by any software except for the TCM. The TCM is also the only authorised software to restore the last valid checkpoint after any power failure. Run-time enforcement of memory protection rules is implemented through replacing, during compilation, all potentially unsafe instructions of untrusted software by hyper-calls, traps within the MPI's hypervisor that check the safety of the executing instructions at run-time. An unsafe instruction can be any control-transfer (call/jump) or memory-access (read/write) instruction with a dynamic addressing mode. Further details can be found in Chapter 3. are illustrated later. To this end, the only requirement for MPI is a few free kilobytes of the non-volatile memory of the underlying device to store its TCM.

For the sake of clarity, we only consider the case of having one untrusted software module deployed in the embedded device. However, our approach generalises to support multiple untrusted software modules at the same time. In what follows, we outline our adversary model and then describe the design rationale behind MPI in detail.

4.3.1 Adversary model

We consider a strong remote software-based adversary $\mathcal{A}\mathsf{dv}$ who has full access to the network or potentially presents itself inside the device in the form of malware. $\mathcal{A}\mathsf{dv}$

can eavesdrop on or tamper with traffic on any communication medium supported by the target device. $\mathcal{A}\mathsf{dv}$ can control the entire software state and data that MPI does not explicitly protect. This includes attempts to inject malicious code, modify any writable memory, read any readable memory, corrupt specific data, or manipulate I/O pins. However, she cannot launch Denial of Service (DoS) attacks. Protection against DoS attacks is out-of-scope in this chapter as they do not alter the memory state of the device. Notably, we consider checkpoint corruption or leakage as a potential attack outcome, possibly achieved through traditional run-time exploits. For instance, $\mathcal{A}\mathsf{dv}$ could manipulate a checkpoint via a run-time attack, leading the device into a persistently compromised state. Interestingly, unlike code injection attacks — typically mitigated by immutable code sections — checkpoint images must reside in writable non-volatile memory, making them inherently more vulnerable to such attacks.

We assume that the TCM is correctly installed on the embedded device initially by a trusted party. We also assume that the TCM code is bug-free and does not contain any memory corruption vulnerability.[1] This means that $\mathcal{A}\mathsf{dv}$ cannot bypass any protection rule enforced by the TCM. We rule out all physical and hardware-focused attacks.

4.3.2 Design rationale

MPI consists of two main components: (i) a trusted software module (TCM) that is installed on the device before deployment, and (ii) a modified *untrusted* compiler toolchain that produces safe binary images of untrusted software to comply with the TCM protection policy.

Trusted Computing Module (TCM)

Figure 4.2 compares the MCU memory layout when applying MPI vs. the state-of-the-art proposals, highlighting the main limitations of such approaches.

In MPI, the TCM is a set of software functions initially installed on the device by a trusted party using a physical programming device, i.e. SPI or JTAG. The TCM utilises part of the non-volatile memory, including the bootloader area, acting as a hypervisor that fully manages access to the memory and other resources. MPI extends PISTIS

[1]The bug-free property can be achieved by formally verifying the code before deploying it. The TCM is unlikely to be updated once installed correctly on the device. Therefore, the formal verification step will take place only once. On the contrary, this is hard to realise considering the untrusted software as it is frequently updated or changed, requiring re-verifying the code with each update, resulting in a cumbersome requirement.

components by introducing a new **checkpoint generator**: a two-flavor software function that can generate a checkpoint image for either untrusted software or the TCM module itself.

Notable is that the TCM maintains a client API that conforms to the GlobalPlatform specifications on Trusted Execution Environments (TEEs) [95] to guarantee the memory protection level required. Thus, the interaction with the TCM can only occur through this API that offers a set of valid entry points. It is worth mentioning that each of these entry points starts execution by disabling all global interrupts to ensure atomic execution of TCM functions. Upon return, the interrupts are enabled.

To be installed in the device's memory, the untrusted software has to be deployed over the air and pass through the TCM's Loader/Verifier module. The entire deployment will be rejected if the software contains at least one unsafe instruction. The right part of Figure 4.2 illustrates the layout of the memory map after applying MPI. The MPI memory protection policy is as follows:

- The untrusted software can only execute instructions within its logical domain or from one of the entry points of the TCM. This controlled execution of TCM functions prevents arbitrary execution of chunks of its code, preventing code-reuse attacks [130, 253].
- The untrusted software cannot read from or write to any part of the non-volatile memory (FRAM).[2] It can only perform R/W operations to a dedicated part of the volatile memory (RAM).
- The volatile memory (RAM) is non-executable, it is only for holding data. This can be achieved by preventing the Program Counter (PC) from jumping to that memory area.
- Likewise, the dedicated non-volatile memory area for checkpoint images is non-executable as it is only used for storing data.
- The TCM is subject to no restriction. It can read, write, or execute any part of the device memory.

Intermittent execution of TCM. So far, we did not consider the intermittent operation of the underlying device. Nevertheless, we designed TCM to operate safely under power failure cases due to the implicit hard-coded *checkpoint Generation Calls* (CGC). Such calls are inserted at certain places where saving progress is important to

[2]If needed, it can only read a portion of its code memory reserved for application data and Memory Mapped Input/Output (MMIO) registers.

maintain the forward progress property of MPI (performance-wise) and prevent unsafe process switching (i.e. giving control to untrusted software) when reviving the execution after a power failure (security-wise). For instance, while receiving software updates, a CGC is triggered after each write of a memory page (a total of 256 bytes). Also, a CGC is triggered with every verification of a set of instructions (in our implementation, we adjusted the number to 128 instructions). It is worth mentioning that the checkpoint memory storage area of MPI is different from the one reserved for the untrusted software. MPI saves its checkpoint images in the TCM memory.

4.3.3 Checkpoints generation and restoration

In this work, we target a broad range of batteryless devices, which even do not have an energy monitoring circuitry used to anticipate the exact time of power failures. MPI entails providing an API along with its enabled toolchain to software developers to simplify the generation of checkpoints. They can invoke the dedicated function to generate checkpoints when needed in their software. Upon invocation, MPI generates the checkpoint and maintains its confidentiality and integrity. MPI also takes the necessary actions to restore program execution from the last valid checkpoint, requiring zero effort from software developers when developing their intermittent software.

Checkpoint generation

As previously mentioned, the TCM maintains a set of valid entry points (the API), using which the untrusted intermittent software can execute some functions located inside the secure memory area. One of these entry points is the checkpoint generation function (*generateCheckpoint* in Figure 4.2), which can be executed by the untrusted software to create a snapshot of its intermediate state. This function backs up the contents of the execution stack, heap, global variables, and general- and special-purpose registers, by atomically[3] copying their contents from the volatile memory or registers to a secure non-volatile memory area (only accessible to the TCM). The size of the non-volatile memory reserved for the checkpoint images is adjustable and application-dependent.

Double-buffering. To avoid corrupting the last valid checkpoint while generating a new one due to the possible power failure, MPI employs a double-buffering technique with a two-phase commit operation. In particular, MPI maintains two independent and

[3]To maintain the memory protection policy, all of the TCM's functions execute atomically by disabling all global interrupts at the beginning of the execution and enabling them upon return.

equal memory areas for storing the checkpoint images. MPI also maintains a 2-bit buffer index (with 0/1 values) dedicated to these areas. MPI sets the index bit of the memory area that holds the last valid checkpoint to 1, and it sets the bit of the other one to 0. When generating a new checkpoint image, the corresponding function inside the TCM will overwrite the area whose index is 0. Then, this function will toggle the index bits (in the 2-bit buffer index) atomically. This operation updates the location of the last valid checkpoint. Power failures up to this point might interrupt the checkpoint generation process and lead to an invalid or partially updated checkpoint. However, since the buffer indexes will be untouched, the memory area holding the last valid checkpoint will be unmodified, and MPI can recover the system from the latest consistent checkpoint. Please note that MPI considers the same double-buffering mechanism when generating implicit checkpoint images when its TCM module executes.

Checkpoint restoration

Our approach entails editing the linker script to include the entire bootloader memory in the TCM memory. It also considers erasing the original content of the bootloader section and replacing it with an initial code (a trusted bootloader code), in which the first instruction to execute is always a fixed one inside the TCM memory. This strategy simplifies the checkpoint restoration process: whenever this initial code is executed with the pre-verification/loading step, it invokes the checkpoint restoration function (*restoreCheckpoint* in Figure 4.2) to restore the computational state using the last valid checkpoint by initializing the stack, registers, and other global variables with their corresponding values stored in that checkpoint.

The checkpoint generator function maintains a persistent binary variable stored in the TCM memory to distinguish the last valid checkpoint image for untrusted software from the one for the TCM. This variable is set to 1 when generating a checkpoint image for untrusted software. It is flipped to 0 when generating a checkpoint image for the TCM. This variable is always set to 1 when exiting the TCM memory and switching the context to the memory of untrusted software. When the *restoreCheckpoint()* function is invoked, it checks the value of this variable. If it is 1, it restores the last valid checkpoint of untrusted software, and accordingly, execution continues starting from the restored value of the program counter (PC). If the value of the binary variable is 0, the last valid checkpoint images of untrusted software and TCM are both restored.[4] In this case, the execution continues considering the PC value of the TCM checkpoint image. Upon

[4]Each of them maintains a separate volatile data memory as shown in Figure 4.2.

return from the TCM memory, the execution continues according to the PC value of the restored checkpoint image of untrusted software.

Each context switch from the insecure world (the untrusted software memory zone) to the secure world (the TCM memory)—for executing other services rather than generating a checkpoint—should start by generating a checkpoint image for untrusted software. This operation is crucial to maintain a consistent state of the device memory before and after a power failure. During execution, the executed service triggers an implicit TCM checkpoint generation call. This does not include virtual instructions as they are frequently invoked, and thus checkpointing will degrade the performance.

4.4 Implementation

We implemented MPI targeting MSP430FR [239] series ultra-low-power MCUs from Texas Instruments, which are the de-facto MCUs for batteryless devices [113, 181, 74]. To the best of our knowledge, these MCUs are the only ones that comprise a combination of SRAM and non-volatile FRAM [240].

MPI in numbers. As aforementioned, MPI consists of a TCM and a GCC compiler plugin for the MSP430 architecture. We implemented the TCM is in the C programming language, except for virtualised instructions and checkpoint generation/restoration functions that we implemented in assembly. In particular, the TCM is composed of 1438 lines of C code along with 414 assembly instructions. We implemented the virtualisation of 28 unsafe MSP30 instructions (out of 99 instructions) by using 354 assembly instructions. The total size of the TCM does not exceed 14 kB. The compiler plugin consists of 1481 lines of Python code, which is invoked transparently by a Makefile consisting of 120 lines of GNU commands. Besides, a custom linker script that includes 530 lines of code, of which 70 lines of code are added to enable MPI. To produce an MPI-compliant binary image, the programmer needs only to adjust some configurations in the Makefile, i.e. include the path of the original GCC compiler toolchain in the underlying machine, add a path to the application source files, etc. and then execute it. Afterward, the programmer can deploy the binary image over-the-air on the target batteryless platform.[5] It is worth mentioning that the TCM is first deployed on the same machine by a trusted party using a physical programming device and custom configuration files. These configuration files include a Makefile consisting of 143 lines

[5]Note that we remain agnostic about the communication protocol as this is usually supported by another MCU.

of GNU commands in addition to a custom linker script consisting of 828 lines of code, of which 368 lines are added to support MPI.

4.5 Evaluation of MPI

In this section, we evaluate MPI to investigate its performance overhead and security guarantees compared to the state-of-the-art secure intermittent systems [93, 231, 76].

Experimental setup. We used TI MSP430FR5969 MCU [238] to perform our experiments. This MCU was mounted on a TI MSP-EXPFR5969 evaluation board, featuring 128 KiB of FRAM and 2 KiB of SRAM. The MCU frequency can be up to 16 MHz. However, in our experiments, we used the default clock rate, which is 8 MHz. We used and modified GNU GCC v8.3.1.25 as a compiler toolchain. The support of Memory Protection Unit (MPU) and hardware-based cryptographic primitives is one of the main factors for selecting this MCU, which allowed us to experimentally compare MPI with the state-of-the-art approaches. When evaluating MPI, we have disabled these hardware modules. We emulated power failures by periodically resetting the MCU using timer interrupts, for the repeatability of the comparative measurements. We used the Energy Trace tool [124] from TI to measure the energy consumption of the benchmarking applications, and a logic analyser to measure the run-time overhead. We depended on the GNU ELF tools to report on the memory footprint.

Benchmarking applications. Due to the lack of a standard benchmark, we consider various intermittent applications that provide a good balance between Input/Output intensive and computationally intensive tasks. The description of these applications is as follows:

- **Sense-Hash:** An application that sends sensed temperature values along with their hash value, computed using a lightweight implementation of SHA2 primitive.
- **Sense-Crypto:** An application that stores sensed temperature values encrypted in memory using a simple encryption mechanism based on the XOR operator, where the secret key is derived based on the length of the data.
- **BitCount:** A computation-based application that counts bits in a random string using several different methods.
- **AR:** An Activity Recognition application that recognises whether the object is moving or in a stationary mode. This application starts with a training phase by

Table 4.2: The overhead incurred by MPI on the execution time of reference applications.

Application	Execution Time (ms) without MPI	Execution time (ms) with MPI	Relative Overhead
Sense-Hash	33.97	58.78	73.03%
Sense-Crypto	71.85	73.56	2.37%
BitCount	33.87	61.48	81.51%
AR	472.7	660.6	39.75%
Storage R/W	7.9	11.17	41.39%
Avg Overhead			**47.61%**

collecting the relevant data in both modes.

- **Storage R/W**: An application that reads big chunks of data from several sources, i.e. RAM, I/O pins, etc., and writes it to the non-volatile memory.

4.5.1 Run-time overhead

As previously explained in Section 4.3.2, MPI is a pure software approach that verifies the safety of some instructions at run-time without any hardware support. This situation introduces some run-time overhead. Table 4.2 illustrates the execution time of all reference applications with and without MPI. With MPI, the run-time overhead amounts to 2.37% in the Sense-Crypto application, whereas it is up to 81.51% in the BitCount application. On average, the execution time overhead of MPI is around 47.61%. The significant divergence of the relative overhead in each application mainly depends on the number of instructions with dynamic addressing modes that have to be checked at run-time. Please note that, while this performance overhead seems high, the overall energy consumption of MPI is comparable to the state-of-the-art approaches while maintaining stronger security guarantees.

4.5.2 Memory footprint

MPI inserts instructions (e.g. special canaries) to the application binary during the instrumentation phase. Therefore, it is crucial to evaluate the impact of MPI on the binary size. Table 4.3 compares the binary size of each benchmarking application when compiled using a standard toolchain with the binary size when considering the MPI-enabled toolchain. The relative size overhead for most of the applications is small, averaging at 7.42%. The aforementioned size of MPI-enabled binary images does not include the size of MPI's TCM itself. The TCM has a fixed size of 13948 bytes (less

than 14 kB). This size along with the extra size that is added to any application is less than any memory footprint reported for any secure intermittent proposal [76, 231, 93]. For instance, the work in [231] requires around 20 kB of additional free memory when the compiled code is optimised for space. The required additional space can grow up to 36 kB when no optimisation flags are used.

4.5.3 Deployment overhead

In principle, the deployment of any application is performed in two steps. First, the application is transmitted over the air to the embedded device as a stream of packets. Second, the receiver code writes these packets to the expected memory part, and then a simple checksum (CRC) function is performed to verify that all packets have been received correctly.

In our evaluation, we have connected the experimental platform to a laptop (software provider) via serial communication at a speed of 9600 bps. Table 4.4 shows the time required to deploy each of the reference applications in both a standard and MPI-enabled mode. The MPI-enabled mode consumes more time in deploying applications due to (i) the extra size of binary images and (ii) the load-time verification done on the device itself, which verifies the safety of the received binary image after performing the CRC check. The average deployment time overhead is not more than 19%.

It is worth mentioning that the deployment process in MPI-enabled devices can be easily extended to be fully secure, maintaining the integrity and confidentiality of the transmitted binary image. Considering that the secure memory area of MPI cannot be read or written, a secret key and some cryptographic primitives, e.g. decryption and Message Authentiation Code (MAC) functions, can be deployed as a part of the TCM during the initialisation phase, which should be performed by the trusted software provider. Upon then, the software provider that shares the same secret key can easily

Table 4.3: The memory footprint of reference applications with and without MPI.

Application	**Binary Size (B) without MPI**	**Binary Size (B) with MPI**	**Relative Overhead**
Sense-Hash	10956	13288	21.29%
Sense-Crypto	5556	5596	0.72%
BitCount	8696	9648	10.95%
AR	16592	17228	3.83%
Storage R/W	5404	5420	0.3%
Avg Overhead			**7.42%**

send an encrypted binary image along with its MAC value to the embedded device. On the embedded device side, the TCM is required to verify the MAC value of the received binary image and then decrypt it before running the load-time verification module. This approach is compliant with SUIT [175], a new IETF standard for software updates for low-end Internet of Things devices, which identifies data confidentiality, authenticity, and integrity as three security properties that must be fulfilled to secure the code update process [13].

Notable is that while the deployment time has been evaluated considering a continuous power-on state, the deployment process in MPI can also be performed intermittently, as explained in Section 4.3.2. The recorded deployment time in Table 4.4 also includes the time used to generate several checkpoints in several places in the code. The idea is that the receiver code is configured to receive each packet, i.e. a memory page, write it in the non-volatile FRAM, generate a checkpoint, and then send an acknowledgment to the software provider to send the next packet. The software provider will always start by sending the related metadata of the binary image that includes among other things the size of the binary image and the number of packets that should be transmitted. If the acknowledgment is not received within the expected amount of time, the software provider re-transmits the last packet again. Once all packets have been received, The TCM starts the verification process that will verify each received instruction while generating checkpoints at certain places to maintain the forward progress property in the case of power failure. As aforementioned in Section 4.3.2, MPI generates a checkpoint after verifying all instructions in one memory page (in the 16-bit MSP430 architecture, the page size is 256 bytes. Thus, there exist 128 instructions in each memory page). The number of verified instructions before generating a checkpoint can be adjusted according to the requirements.

Table 4.4: The relative overhead of deployment time of reference applications with and without MPI.

Application	**Deployment Time (ms) without MPI**	**Deployment Time (ms) with MPI**	**Relative Overhead**
Sense-Hash	10448	14281	36.54%
Sense-Crypto	5872	6669	13.53%
BitCount	8522	10555	23.85%
AR	15638	16918	8.18%
Storage R/W	5672	6310	11.2%
Avg Overhead			**18.66%**

4.5.4 MPI vs. other intermittent approaches

We experimentally evaluated the performance of MPI vs. other existing approaches that target secure intermittent systems. For a fair comparison, we implemented the same checkpoint generation and restoration functions in both MPI and MPU-dependent, i.e. [76] approaches. We extended the aforementioned functions with encryption/decryption capabilities, leveraging the existing AES hardware module with a 128-bit secret key, to compare with the encryption-based approach, i.e. [93]. We did not include the hash-based approach [231] in the comparison due to the need for a tamper-free memory that does not exist as a standard hardware module in any intermittent computing platform. This approach represents a middle ground between the two other considered approaches. Furthermore, a systematic comparison with MPI is provided in Table 4.1.

Checkpoint generation time

Table 4.5 illustrates the time and energy consumed in MPI and the other approaches when generating a checkpoint. Whenever executed, the checkpoint generation function had to back up a stack of size 800 bytes along with all general- and special-purpose registers.[6] It is clear that the encryption-based approach is time- and energy-consuming, whereas MPI performs slightly better than the MPU-based approach. This is due to the extra energy that is consumed by the MPU registers. Due to the limited space, we did not consider comparing results when executing the checkpoint restoration function as the relative performance will be the same.

End-to-end analysis

While MPI performs better than any existing secure intermittent approach when generating a checkpoint, we performed an end-to-end evaluation since MPI-enabled ap-

[6]Please note that the entire allocated SRAM for any application is 1500 bytes. The longest stack built by any application has never been more than 800 bytes.

Table 4.5: Time and energy consumed for a checkpoint generation in MPI vs. other approaches.

Approach	**Checkpoint generation (832 B)**	
	Time (ms)	Energy (μJ)
MPI	1.206	7.791
MPU-based, i.e. [76]	1.2066	9.181
Encryption-based, i.e. [93]	4.93	19.757

plications incur extra run-time, and accordingly, energy overhead. We performed a comparison between MPI and four other approaches by considering the BitCount application to show the worst-case scenario. The selection of the BitCount application is due to the worst run-time overhead considering MPI. We configured the application as a single task, which asks the TCM for generating a checkpoint every 40 ms. We also configured an artificial periodic power failure through interrupts every 50 ms. The selection of this value is mainly to force power failure before the completion of the BitCount task which takes around 61.48 ms as reported in Table 4.2.

Table 4.6 shows the consumed energy and the number of tasks that have been achieved for 5 seconds with various approaches, including MPI. We implemented the insecure intermittent using the same checkpoint generation and restoration functions without considering any security measure. When considering no optimisation (-O0) during the compilation process of the BitCount application, MPI performs worse than any listed approach in terms of the energy consumption and the number of achieved tasks within five seconds. This situation happens since all instructions with dynamic addressing mode are left untouched. All of these instructions need to be instrumented, which incurs a high run-time overhead and affects the number of achieved tasks. It is clear that the "No Checkpoint" approach performs better than MPI due to the less execution time needed to complete the task of the BitCount application when compared to the execution incurred by the MPI-enabled BitCount version.

However, when considering the fast optimisation option (-O3), the compiler reduces the number of instructions with dynamic addressing mode by re-writing them in a static nature. This situation enhances the execution time and also the number of executed tasks in all approaches. However, the increase of the achieved tasks by MPI is higher than in any other intermittent system considered due to the elimination of many run-time checks that do not exist in other systems. Such run-time checks are no longer needed as their corresponding instructions are re-written statically by the compiler (when using -O3). This means that load-time verification is enough to ensure the same level of security. For instance, Table 4.6 shows that the number of tasks that MPI has achieved is higher than the number of achieved tasks by the default No-checkpointing approach as well as the encryption-based one. The MPU-based approach performs slightly better in terms of the number of achieved tasks, although it consumes more energy than MPI.

On the other hand, if we select another application with a low run-time overhead, MPI performs better than any existing secure intermittent proposal. Considering the Sense-Crypto application, Table 4.7 shows that, MPI achieves the same number of

Table 4.6: End-to-end analysis of energy consumption and throughput of BitCount application when using MPI vs. other approaches, considering a period of 5 seconds, where a power failure occurs every 50 milliseconds.

Approach	Throughput/Energy for 5 seconds (BitCount)			
	No Optimisation (-O0)		Fast Optimisation (-O3)	
	Energy (mJ)	Task	Energy(mJ)	Task
No Checkpoint	21.390	82	26.066	286
Insecure Checkpointing	22.446	116	26.476	377
MPU-based, i.e. [76]	22.958	92	26.980	371
Encryption-based, i.e. [93]	22.781	115	26.736	132
MPI	23.226	67	26.498	351

Table 4.7: End-to-end analysis of energy consumption and throughput of Sense-Crypto application when using MPI vs. other approaches, considering a period of 5 seconds, where a power failure occurs every 50 milliseconds.

Approach	Throughput/Energy for 5 seconds (Sense-Crypto) No Optimisation (-O0)	
	Energy (mJ)	Task
No Checkpoint	26333	0
Insecure Checkpointing	26843	66
MPU-based, i.e. [76]	26967	65
Encryption-based, i.e. [93]	28569	55
MPI	26882	65

tasks as the MPU-based approach, consuming less energy and without considering any optimisation technique. It is clear that the "No Checkpoint" approach does not achieve any task for the selected application because the needed execution time(i.e. 71.85 ms) is higher than the time interval set for a power failure (i.e. 50 ms). Notable is that the security of MPI is superior compared to all existing techniques as we clarify in the next section.

4.5.5 Security evaluation

The main goal of MPI is to provide unconditional integrity and confidentiality guarantees to checkpoint images in embedded devices without depending on any hardware assumption, i.e. the existence of some hardware support. MPI provides such security guarantees depending on its Trusted Computing Module (TCM). Any memory corruption vulnerability in the TCM hinders the realisation of such security guarantees. Therefore, considering that the TCM is a software module of a reasonable size (less

than 2000 lines of code), developed to be accommodated in a wide range of devices, formally verifying the memory safety properties of this module before deploying it is possible, incurring only some development overhead for one time. High profile examples of trusted execution architectures that have considered formally verifying the implementation of their modules include VRASED [185] and the MicroVisor [15]. This makes MPI secure against all software-based remote only attacks (Section 4.3.1). This level of security is not guaranteed in other existing intermittent proposals, i.e. [76, 93, 231], that target the security of checkpoints, due to the dependency on normal applications that might be buggy or malicious. Furthermore, software developers cannot formally verify their applications as this would require continuous verification efforts with each update per application. This is a costly requirement to meet. In short, MPI is the only intermittent computing approach that is secure against all malware-infestation attacks, providing strong integrity and confidentiality guarantees using a lightweight pure-software hypervisor approach.

From the system point of view, in contrast to all existing approaches, MPI is an error-free approach, as it does not require any intervention from software developers to do any hardware or software configurations. This significantly reduces the burden on software developers when implementing their software modules. For instance, in SIA [76], developers have to deal with very complicated application-dependent configurations to activate the Memory Protection Unit (MPU). In general, the design of MPUs has some shortcomings that limit their usage in practice [271]. Furthermore, the use of MPU assumes that the malicious software resides in a non-privileged memory area and thus cannot tamper with the MPU configuration at run-time. However, this assumption requires a trusted compiler toolchain to prevent any non-privileged malicious code, i.e. statically-linked libraries, from tampering with, i.e. overwriting, the MPU configurations during the compilation phase. In MPI, the compiler toolchain is untrusted.

4.5.6 Flexibility and portability

Compared to hardware-based solutions, which require detailed configuration that is rigid and error-prone, MPI does not dictate any specific coding style since there is no programmer intervention. The programmer is oblivious to the underlying protection mechanism since the toolchain provides the necessary code instrumentation for checkpoints protection. MPI can work on any intermittent computing platform that employs checkpoints and comprises non-volatile memory, and it does not depend on a specific

microcontroller, e.g. MSP430FR. Since it is a purely software-based solution, it can be ported to other microcontroller-based computing platforms. This feature makes MPI support even future computing platforms that does not exist now but will appear in the market. On the other hand, likewise PISTIS [103], MPI can also be deployed on battery-powered devices to provide basic security features such memory isolation.

4.6 Summary

In this chapter we proved the flexibility of software-based security solutions by addressing a security challenge in intermitted computing. In particular, we presented MPI, a lightweight software-based security-oriented checkpoint system well-suited for devices that cannot count on a reliable power source. In particular, we showed how applications in these harsh environments can ask MPI for the generation of a checkpoint, from which the application will be able to seamlesly resume its operation upon a power outage. MPI provides strong confidentiality and integrity guarantees of the checkpoint images and other sensitive data in the presence of a strong software-based remote-only adversary. Although the intermittent computing is a challenging domain due to the lack of a stable power source, we showed how software-based approach can provide solid security guarantees. Our evaluation results further showed that MPI incurs a reasonable performance overhead, which makes it suitable for the entire spectrum of batteryless platforms.

Chapter 5

FLAShadow: A Flash-based Shadow Stack for Low-end Embedded Systems

This chapter is based on a work previously published as:

Grisafi, M., Ammar, M., Roveri, M., Crispo, B. (2024). FLAShadow: A Flash-based Shadow Stack for Low-end Embedded Systems. ACM Transactions on Internet of Things, 5(3), 1-29.

For the sake of conciseness and to avoid excessive redundancy, some parts of the original publication were adapted to better fit the thesis structure. We invite the reader to refer to the original publication for more details.

Preamble

In the previous chapters, we have seen how a software-based TEE is capable of providing strong security guarantees to the low-end embedded systems. We have provided examples of different use cases and security services that are supported by PISTIS, e.g. Remote Attestation and Secure Code Update. These can be seen as Trusted Applications (TAs) which, traditionally, play a passive role in a system. In particular, they are typically invoked by an application, or by the TEE itself, after a specific event (e.g. a request from a remote party). If, on the one hand, this type of security services provide good security guarantees, on the other hand, they are better at detecting and reacting to security threats rather than preventing them. There are other types of security services which are instead more active and intrinsically connected to the target application that needs to be protected. Some can be seen as actual security primitives, e.g. memory

isolation, while others can be seen as active security services. In this chapter, we focus on the latter, and we present FLASHADOW, a Flash-based Shadow Stack for low-end embedded systems.

Run-time attacks are a rising threat to both low- and high-end systems with the spread of techniques such as Return-Oriented Programming (ROP), which aims at hijacking the control flow of vulnerable applications. Although several control flow integrity schemes have been proposed by both academia and the industry, the vast majority of them are not compatible with low-end embedded devices, especially the ones that lack hardware security features. Shadow stack is a form of control-flow integrity protection that ensures that attackers cannot hijack a vulnerable application by tampering with the return addresses stored on the stack. In this chapter, we will explore the impact that such a solution has on both the security of an application and on its performance. Furthermore, we will see how FLASHADOW can be implemented on a low-end embedded system even in the absence of hardware security features, simply leveraging software. This will be yet another step towards the comprehensive protection of low-end embedded systems. We evaluate an open-source implementation of FLASHADOW for the MSP430 architecture, showing an average performance and memory overhead of 168.58% and 25.91%, respectively. While the average performance overhead is considered high, we show that it is application-dependent and incurs less than 5% for some applications.

Chapter outline. The remainder of the chapter is organised as follows. We will present some background on the motivation behind this work in Section 5.2. We will then briefly discuss how PISTIS was extended in Section 5.3, and then focus on the details of FLASHADOW in Section 5.4. Implementation details and evaluation are reported in Section 5.5 and Section 5.6 respectively. Section 5.7 discusses some possible extension of FLASHADOW and finally, Section 5.8 reviews the related work and shows how FLASHADOW is a first of its kind.

5.1 Introduction

In recent decades, computer systems have been evolving on all fronts, aiming at providing better experiences for users. Nevertheless, the foundation of all software stacks in most systems is still written in unsafe programming languages such as C and C++ for obvious flexibility and performance reasons. Memory corruption vulnerabilities are very common in such languages because they require the programmer to manually manage

memory allocations. This opens the door for committing programming mistakes that result in vulnerabilities, which can be exploited in several ways. Due to the widespread deployment of defences such as Data Execution Prevention (DEP) [173] that have raised the bar for successfully mounting traditional code-injection attacks [233], control-flow hijacking attacks have become the most popular software exploitation technique over the past years. Such attacks, exemplified by Return-oriented Programming (ROP) [211] and Jump-oriented Programming (JOP) [39], circumvent traditional deployed defences, e.g. DEP and Address Space Layout Randomisation (ASLR) [2], by reusing some code snippets, called *gadgets*, that exist in memory.

In particular, control-flow hijacking attacks exploit memory corruption vulnerabilities, e.g. buffer overflows, use-after-free, etc., to divert program execution from its intended control flow by executing a chain of gadgets in a controlled manner. Hence, rather than injecting malicious payloads directly onto the stack or heap, where DEP mechanisms block it from being executed, control-flow hijacking attacks intend to inject addresses of existing in-memory code fragments, so-called *gadgets*, onto the victim stack (i.e. ROP attacks [211]) or heap (i.e. JOP attacks [39]), causing the victim application to execute its own binary code in an unanticipated order. As the names indicate, ROP attacks target backward edges by means of gadgets ending with return (`RET`) instructions, whereas JOP attacks target forward edge gadgets that end with indirect jump or call (`JMP/CALL`) instructions.

To narrow the focus, ROP is the most widely leveraged attack technique, posing a higher significant threat, compared to other control-flow hijacking attacks, for both high-end devices [223, 34] as well as embedded systems [46, 141, 160, 3]. It is also expected that ROP attacks will increase in frequency and are still dangerous despite the deployment of some defences [127, 43, 44]. Mitigating ROP attacks requires guaranteeing the integrity of return addresses when pushed onto the stack. So far, a multitude of defences have been proposed by both industry and academia to prevent control-flow hijacking attacks, most notably Control-Flow Integrity (CFI) mechanisms for both forward [1, 268, 244, 236] and backward [269, 43, 68, 152] edges. CFI guarantees that program execution follows one of the valid paths in a pre-generated static Control-Flow Graph (CFG). Forward-edge CFI techniques are deployed by big IT players such as Google to protect Chrome and Android [241], and Microsoft to protect Windows 10 and other higher versions [172]. On the contrary, the most comprehensive backward-edge CFI mechanisms such as safe stacks [242] and shadow stacks [243] have never seen wide adoption despite being available in mainline compilers. This is mainly because they can be easily bypassed via information disclosure attacks that would leak the loca-

tion of the shadow/safe stack, thus tampering with its content [43, 44]. Additionally, the performance cost of some implementations on some architectures is not tolerable [68].

Yet, regardless of the aforementioned limitations, existing ROP defences are not compatible with many of the deployed low-end embedded devices due to one or more of the following reasons. First, the underlying MicroController Units (MCUs) of embedded systems have constrained resources, e.g. a few KBs of Flash and RAM, and lack basic protection features found on high-end desktop systems. For instance, the vast majority of corresponding MCUs do not have Memory Management Units (MMUs) to enable DEP and ASLR protection mechanisms that are needed as a prerequisite for any ROP defence. Second, a significant number of embedded systems run bare-metal applications, lacking any Operating System (OS) that would help in enforcing security policies. Third, MCUs load and run a single statically-linked binary image in a single address space, making it very challenging to compartmentalise any part of the software securely. Notably, statically linked binaries do not decrease the attack surface for ROP attacks, possibly leading to the presence of additional ROP gadgets.

As a countermeasure against ROP attacks on embedded systems, several directions have been followed by the research community. For instance, various control flow attestation schemes have been proposed to detect control-flow hijacking attacks on embedded devices by a trusted remote party [4, 75, 186]. Such schemes can only detect attacks after happening with no means of providing prevention capabilities. Another direction was to enforce backward edge integrity through some form of CFI schemes that would suit embedded systems [11, 77]. More recently, a shadow stack approach has been proposed for securing return addresses in embedded devices [270]. Despite the potential advantages that distinguish each of the aforementioned techniques, all of them depend on hardware features that might not exist in all embedded devices or are basic enough to be easily disabled by a motivated attacker [102]. This brings up two important questions, which are: (i) Is it possible to secure embedded systems with zero hardware security features against ROP attacks? and (ii) if so, would the proposed solution be secure enough and acceptable in terms of performance and memory overhead? To fill this knowledge gap, this chapter proposes and evaluates FLASHADOW, a shadow stack approach for low-end embedded systems, requiring no hardware security features.

In doing so, we address a security issue on many of these legacy devices that lack proper security hardware, thus representing vulnerable potential targets for ROP attacks. Notably, such devices, spanning several architectures such as MSP430, Arm and AVR, are still widely employed and will most likely not be replaced in the foreseeable future.

FLAShadow extends and builds atop PISTIS [103], a pure software memory isolation technique, to fully protect the shadow stack from malicious tampering. In particular, PISTIS is a software-based Trusted Computing architecture that offers basic security features such as memory isolation, data execution prevention, and remote attestation, through selective software instrumentation and load-time verification of binaries. To guarantee the integrity of return addresses, FLAShadow dedicates a write- and execute-protected memory area in the Flash, to which a copy of return addresses is pushed and then popped by inline instructions that are inserted and verified by the trusted architecture. The access control rules are also enforced by our architecture, where the adherence to them is verified on the device itself, eliminating the trust of the compiler toolchain. To achieve full protection, FLAShadow leverages some of the PISTIS's security features to complement its role. For instance, the static Remote Attestation (RA) module is leveraged to guarantee the integrity of the loaded binary onto memory. Unlike the design and implementations of software-based shadow stack in high-end systems [68, 243], FLAShadow is secure against information leaks and can be deployed on low-end systems without the need for any randomisation scheme. The average memory overhead of FLAShadow is centred around 25.91%. While the performance overhead might not be tolerable for some applications (up to 578.16%), we show that FLAShadow incurs less than 5% of performance overhead for others.

In a nutshell, this chapter makes the following contributions:

- A design of a novel Flash-based shadow stack technique for generic low-end embedded systems that lack hardware security features.
- An open-source implementation for the MSP430 architecture, publicly available at [106].
- An extensive evaluation of the proposed design on different applications, showing the impact on performance, memory, and energy consumption.

5.2 Preliminaries

5.2.1 Return-Oriented Programming (ROP)

ROP is a classical exploit technique that allows an attacker to execute code in the presence of security defences that prevent injecting shell code. The idea behind this exploit is that the attacker exploits a memory corruption vulnerability to gain control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences, called *gadgets*, which are already present in the code memory of

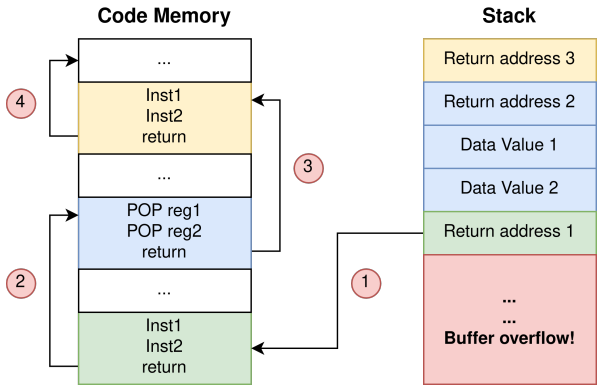


Figure 5.1: A visualisation of the steps of Return-Oriented Programming attacks (ROP).

the target device. Each gadget typically ends in a return instruction and is located in a subroutine within the existing program or linked library code. By chaining these gadgets together, the attacker would be able to perform the malicious task. Figure 5.1 visualises the principle behind ROP, assuming a buffer overflow vulnerability that allows the attacker to overwrite a return address. Notable is that ROP attacks or any of its variants (e.g. ROP without return [50]) could succeed in the presence of several combined defences such as stack canaries, Data Execution Prevention (DEP), and Address Space Layout Randomization (ASLR). This motivates the need for other effective defences such as shadow stacks.

5.2.2 Scope of embedded devices

FLASHADOW targets tiny embedded devices that have small MCUs based on the Von Neumann architecture with little or no hardware security features. In general, such MCUs have a single core and feature Flash and SRAM memories. They execute instructions in place (in physical memory) and have no Memory Management Unit (MMU) to support virtual memory. Some of them can support a memory protection unit (MPU). However, our design and implementation neglect the existence of MPUs due to their shortcomings as clarified in [271] and [102]. In particular, FLASHADOW targets single-thread, yet multi-tasking bare-metal applications that are the most common ones in the IoT domain [59]. Notably, there are still many of these low-end devices employed in both critical and non-critical fields, due to their low cost and power consumption. However, due to their poor hardware configuration, they are incompatible with most state-of-the-art security techniques. Consequently, they are often unprotected against common threats such as ROP attacks, which, depending on the scenario, could lead to

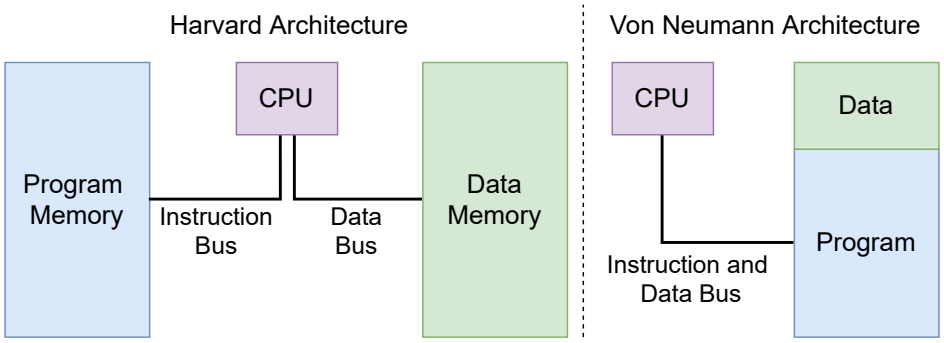


Figure 5.2: Harvard vs. Von Neumann architecture.

disastrous consequences.

Our implementation is based on the MSP430 architecture. This choice is due to the wide use of this architecture in many IoT devices as well as research prototypes. Nevertheless, our design has the same requirements as PISTIS [103], and it is thus applicable to other low-end MCUs in the same class, such as Arm Cortex-M.

5.2.3 Challenges in designing FLAShadow

To guarantee security, any target system should be able to protect the building blocks of the provided security services against malicious tampering. This is especially important when software is the backbone of these services. Any kind of protection cannot be enforced without some sort of *isolation*. Trusted Execution Environments (TEEs) are proposed to fill this gap. However, they have been realised in mid-range and high-end devices. The low-end spectrum of embedded devices lacks rich hardware security features that provide memory isolation in addition to confidentiality guarantees. To tackle this issue, we first propose PISTIS, a run-time-based memory isolation mechanism that depends on software instrumentation. Atop it, FLAShadow is constructed as an effective pure-software shadow stack approach to mitigate ROP attacks. The design of both PISTIS and FLAShadow is not straightforward and has several challenges to handle, most of which have been addressed in Chapter 3.

Still, realising FLAShadow on top of PISTIS poses another challenge as hosting the shadow stack in either the RAM or Flash has pros and cons. While the RAM can be enforced by PISTIS to be non-executable and thus appears as a good choice for FLAShadow, it is very limited in terms of size and would incur high performance overhead to fully protect it. On the other hand, implementing FLAShadow in part of the Flash requires a very sophisticated design that should consider, among many other factors, (i) the execute-only property to ensure strong security guarantees, (ii)

the writing and read mechanisms of the Flash that are more complex, compared to the RAM, and (iii) the right location of the stack to avoid corrupting any functionality when growing in size. We design and implement FLASHADOW on top of PISTIS in part of the Flash, considering the aforementioned challenges. Further details about design decisions are provided in the next sections.

5.3 Memory isolation

Systems security strongly relies on the concept of *trust* as lacking it renders any security service infeasible. Trust is typically assured via some sort of *memory isolation*, a mandatory security primitive for all security services. To this end, we first construct PISTIS as a software-based memory isolation technique, atop which FLASHADOW is built. Our memory isolation primitive follows the bottom-up approach to create an Arm TrustZone-like capability [23] that is highly optimised for low-end embedded devices without hardware support while allowing for DMA and interrupt operations. Compared to Arm TrustZone, our software architecture itself plays a secure monitor role with multiple secure entry points [24]. To achieve so, we initially deploy an initial code, called a Trusted Computing Module (TCM), that occupies part of the Flash memory including the bootloader area. The main goal of the TCM is to guarantee memory protection by isolating its memory area from other memory parts, creating two logically isolated memory zones: secure and insecure. The TCM leverages the secure memory part to deploy security services and primitives like FLASHADOW. Please note that while our memory isolation technique adapts some of the SFI techniques, i.e. binary re-writing, in its special domain, it does not depend on any hardware feature, such as segmentation registers in MPUs as in [112]. Furthermore, in contrast to standard SFI mechanisms [234], we do not trust the compiler toolchain. This significantly broadens the attacker model and boosts the security guarantees of our approach. The correctness of the memory isolation design in PISTIS is formally verified as clarified in [103].

In what follows, we outline the adversary model that is also valid for FLASHADOW, and then describe the design of PISTIS in detail. The design of FLASHADOW is discussed in the next section.

5.3.1 Adversary model

We consider a software-based adversary $\mathcal{A}\mathsf{dv}$ who has full access to the network or potentially presents inside the device itself in the form of malware. $\mathcal{A}\mathsf{dv}$ can eavesdrop

on or tamper with traffic on any communication medium supported by the target device. $\mathcal{A}$dv can also control any software deployed in the insecure memory part of the device. This includes trying to inject malicious code, reading or writing any memory address that is not explicitly protected, corrupting specific data, manipulating I/O pins, or mounting ROP attacks. More importantly, $\mathcal{A}$dv could compromise the toolchain, and tamper with the code before it is loaded on the device. Notably, we consider Denial of Service (DoS) attacks out-of-scope as they do not tamper with the device memory, and the literature proposes several complementary security solutions to address this class of attacks [5] [133].

We assume that the TCM is initially and correctly installed on the embedded device by a trusted party. This can be easily achieved during a new IoT deployment or, in the case of existing systems, it suffices a clean reset of the device with a software update. We also assume that the TCM is bug-free and does not contain memory corruption vulnerabilities.[1] This means that $\mathcal{A}$dv cannot bypass any protection rules enforced by the TCM. Finally, we rule out all physical and hardware-focused attacks.

5.3.2 PISTIS: from the ground up

In Chapter 3 we introduced PISTIS, a software-based memory isolation technique that guarantees memory protection for the TCM and the security services deployed on top of it. PISTIS is designed to be a lightweight and efficient memory isolation technique that can be deployed on low-end embedded devices without hardware support. FLAShadow builds on top of it, slightly modifying some of its components to accomodate the new shadow stack. In this section we will discuss such modification. Please refer to Section 3.4 for further details on PISTIS.

As Figure 3.2 showed, PISTIS offers memory protection to its TCB, preventing any untrusted software from accessing it. We leverage this property to build FLAShadow on top of it, and place our shadow stack inside PISTIS TCB, as shown in Figure 5.3.

Memory isolation: variable length instructions

As explained in Section 3.4.2 tackles the issue of variable-length instructions by inserting special *instruction canaries* in the code. It then ensures that the unstrusted application jumps and returns only where such canaries are present. This is crucial to prevent $\mathcal{A}$dv from bypassing the memory protection enforced by PISTIS. FLAShadow makes some

[1] Likewise the work in [14], the memory-safety property can be achieved by formally verifying the code before deploying it.

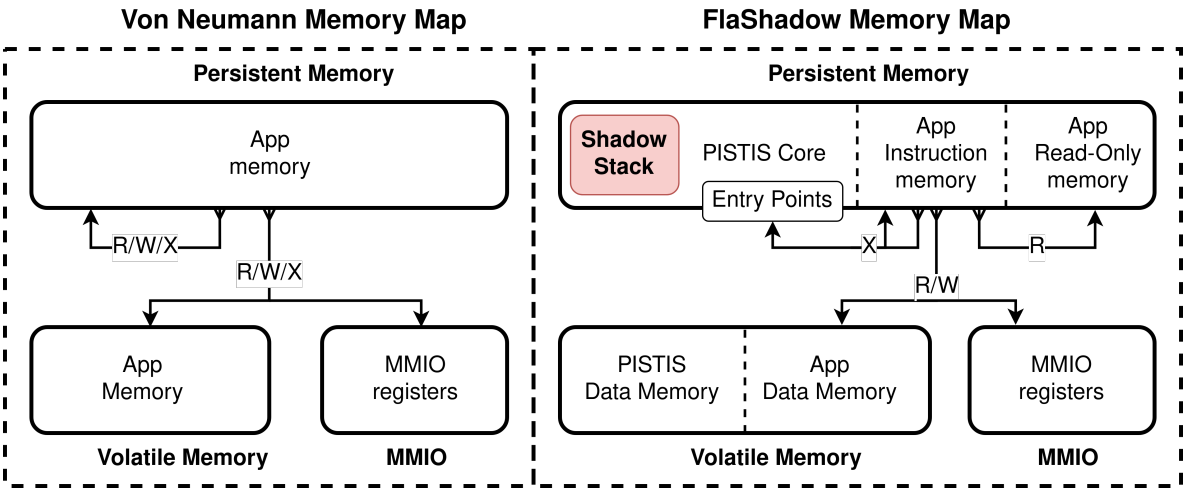


Figure 5.3: A standard memory map vs. the FLASHADOW-enabled memory map in a Von Neumann architecture. The latter also shows R/W/X privileges of the untrusted application.

of this instrumentation superflous by leveraging the shadow stack to secure the return addresses. Therefore, we reduce the number of canaries inserted in the code, only leaving those used for dynamic jumps, as shows in Figure 5.4.

5.4 FLASHADOW

Low-end embedded systems can be particularly vulnerable to advanced attacks such as Return-Oriented Programming (ROP). These aim at bypassing traditional security primitives, such as Data Execution Prevention (DEP), to divert the application control flow of an application and execute arbitrary code. Many embedded systems lack the hardware support to mount adequate defences such as shadow stacks, thus leaving a considerable attack surface for $\mathcal{A}$dv-s.

To fill this security gap and offer protection against these advanced attacks on low-end devices, we propose FLASHADOW, a unique software-based shadow stack design. FLASHADOW contributes to the protection of the application control flow, by preventing $\mathcal{A}$dv-s from tampering with the return addresses on the stack. To the best of our knowledge, FLASHADOW is the first software-based shadow stack solution that does not rely on any hardware component, e.g. MMU or MPU, and offers strong security guarantees. As such, its design is compatible with most low-end embedded systems that are not equipped with rich hardware features. Notably, the lack of such features often impedes the use of most state-of-the-art security solutions, and replacing these devices

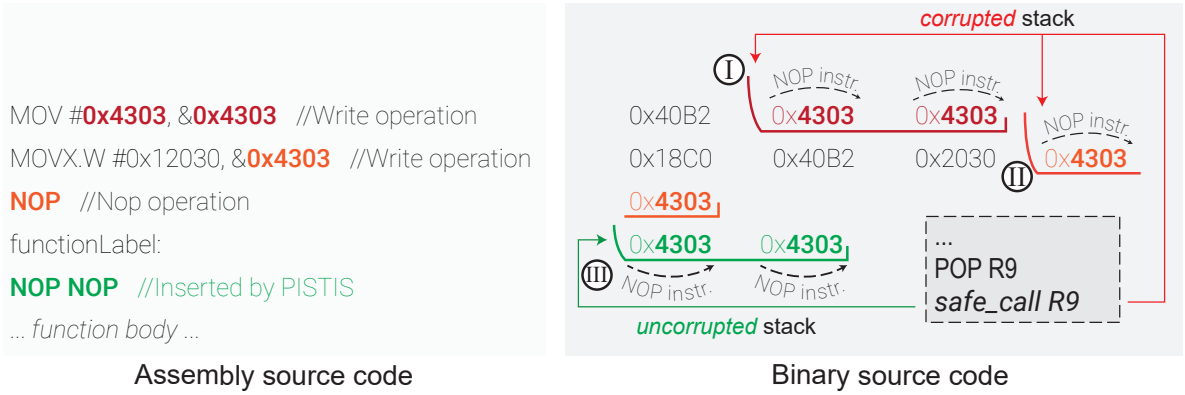


Figure 5.4: Binary code (right) of the assembly source code (left) has 3 NOP slides: a legitimate slide (III) at the beginning of a function and two *accidental* slides (I) and (II) in the middle of two instructions. (III) is inserted by PISTIS. However, (I) and (II) are either maliciously inserted or accidentally derived from combinations of opcodes and operands. The `safe_call` virtual function only allows calls to locations having a slide. For instance, in case of a corrupted stack and a dynamic call, the control flow can at most be diverted to another location containing an accidental slide. However, the instruction executed after the slide will always be safe since verified.

with more capable ones is often impractical and expensive. Conversely, FLAShadow does not require any secure hardware component, thus being a suitable alternative to the lack of security.

Shadow stack is a fully precise technique that verifies the integrity of backward edges with each execution of a return instruction. The main idea is to store a copy of the return address in a separate, isolated region of memory that is not accessible by the attacker. Upon returning, the integrity of the program return address is checked against the protected copy on the shadow stack. Execution is terminated if both values are not equal. As such, FLAShadow has been designed to ensure that return instructions correctly return to their legitimate callers. Nevertheless, this is not straightforward in embedded systems, considering the lack of hardware support.

To compensate for the lack of proper security hardware, we leverage PISTIS, a purely software-based approach and extend its security guarantees with the ones of a shadow stack. As such, FLAShadow is compatible with any device that supports PISTIS and can be deployed alongside it on new devices using any trusted software update technique (e.g. using JTAG or over-the-air (OTA) updates). Once deployed, Flashadow transparently takes care of protecting the untrusted applications that are compiled with our toolchain and deployed with our Secure Update service.

To accomplish so, FLAShadow introduces: (i) an integrity-protected memory area,

(ii) an integrity-protected shadow stack pointer, and (iii) a firmware integration for stack manipulation. The integrity protection is provided by PISTIS, while the firmware integration is built on top of its TCM.

5.4.1 FLASHADOW: design

To achieve its goal, a shadow stack must be secured against unauthorised accesses, e.g. $\mathcal{A}$dv-s trying to inject malicious return addresses. There exist several techniques to protect the shadow stack, one of which considers using a *hidden* memory area through randomisation [68, 243]. However, other than being vulnerable to information disclosure attacks [43], these approaches are not suitable for embedded systems due to the limited entropy. For this reason, FLASHADOW opts for a sounder approach by introducing an integrity-protected memory area to hold a copy of the application return addresses during execution. Notably, the entire area could be leaked to an attacker without compromising the security of the system as $\mathcal{A}$dv-s would need to modify such values.

Although integrity protection is often provided by an MMU- or MPU-like hardware, FLASHADOW leverages PISTIS to achieve so. In particular, we implement the shadow stack on top of the software-based memory protection primitive of PISTIS, without incurring any extra design overhead. In practice, this translates to adding a new protected area to PISTIS AP, as can be seen in Figure 5.3.

In embedded systems, protecting the integrity of the sole shadow stack is not enough to prevent ROP attacks. The integrity of the Shadow Stack Pointer (SSP) needs to be protected as an attacker with access to it could invalidate it, i.e. overwrite it with a pointer to an attacker-controlled memory area. To protect SSP, FLASHADOW considers storing it in a protected register that is exclusively reserved for this purpose. The value of this pointer always points to the top of the shadow stack, so that the system knows which return address to pop upon return, and where to push the new address when executing a call instruction.

Finally, any system using a shadow stack must implement a mechanism to store (push) and fetch (pop) the new protected addresses on the new stack. Although on high-end systems this is accomplished with dedicated hardware, FLASHADOW introduces new software hooks to both push and pop operations. Specifically, we extend the custom toolchain and the verifier module to treat all application `CALL` instructions as unsafe, so that each of them is instrumented. Then, we design a new `safe_call` and `safe_return` virtual function, with a sequence of safe operations aiming to manipulate the shadow stack every time a function is called or returned from.

In summary, FLAShadow comprises a new protected memory area for the return addresses, a new protected register for the safe shadow stack pointer, a compiler/verifier extension for the instrumentation of every application call instruction, and a TCM extension to some of its virtual functions.

It is worth mentioning that FLAShadow also makes some of the original PISTIS features redundant. For instance, the NOP slide mechanism described in Section 5.3.2 is slightly revised. The introduction of a shadow stack makes the NOP slide superfluous for return statements. The new FLAShadow operations replace the `safe_return` NOP slide verification. Nevertheless, this mechanism is still required for forward edges, mainly for dynamic call instructions.

5.4.2 FLAShadow: Flash-based stack

Traditionally, shadow stacks are maintained in the RAM for performance reasons. In light of the revisited AP in Figure 5.6, creating an integrity-protected RAM segment would incur a re-design of the current PISTIS architecture, which leverages the Flash controller to minimise the complexity of its memory protection primitive and thus reduce the execution overhead. Therefore, we design FLAShadow to store the new shadow stack on the Flash memory (Flash + Shadow Stack = FLAShadow), effectively leveraging the native PISTIS TCM without losing its performance optimisation. PISTIS enforces an integrity-protected area over the entire Flash, preventing any unauthorised write operation. This satisfies the FLAShadow security requirements, allowing it to operate without interference from attackers. Notably, the only requirement for FLAShadow is PISTIS, i.e. a TCM with memory isolation, which is entrusted with the protection of the shadow stack. Porting FLAShadow to a RAM-based implementation of PISTIS only requires an engineering effort.

While the use of Flash reduces the complexity of the overall security architecture, effectively minimising the software instrumentation required, it also introduces several design challenges stemming from the intrinsic rigidity of Flash. We recall that a shadow stack is a highly dynamic memory structure that grows and shrinks whenever new values are pushed to or popped from its top. In practice, new return addresses are often pushed to a previously popped location, overwriting the stale content of the shadow stack, thus maintaining the correct call sequence. However, this *overwrite* operation can be quite challenging on a Flash memory, where write operations can only target a cleared memory location, i.e. that has never been used or that has been previously cleared via

a memory erasure operation.[2] This means that whenever a value is popped from the shadow stack, the memory location containing the popped value must be cleared before another value can be pushed to it.

Considering that the smallest area that can be erased in the Flash of any embedded device is a block of bytes (e.g. 512 byte), overwrite operations become cumbersome and particularly expensive to modify a single value, requiring (i) copying its entire segment in RAM, (ii) modifying the value in RAM, (iii) erasing the Flash segment, and finally (iv) overwriting the original Flash segment with the modified segment from RAM. Notably, zeroing a single Flash address, i.e. overwriting it with 0, is a one-cycle CPU operation.

To overcome this limitation, we propose a novel only-growing stack design that does not involve overwrite operations. Starting from an empty Flash segment,[3] we always push new values to the first *free* address at the top of the stack, i.e. the first never-before-used stack entry. To keep track of the popped values, and thus remove them from the stack, we zero them. As a result, at any point in time the stack contains an interleaved sequence of zero (popped) and non-zero (un-popped) values, effectively representing the current sequence of calls.

By never re-using popped locations, FLASHADOW avoids the expensive overwrite operations, thus reducing the run-time overhead. However, this has a considerable impact on the size of the shadow stack. Rather than growing linearly with the depth of the calling sequence, FLASHADOW grows linearly with the total number of function calls. Let us define the size of a stack, at any point in time, as the difference between its base address and the address of its first available slot. Considering the application in Listing 5.1, a traditional shadow stack would never exceed the size of 1: each *return* shrinks the stack size, and each *call* increases it. On the contrary, considering FLASHADOW, the size of the shadow stack is 3, as with each new call, we use a new memory location, no matter if the other locations are no longer used.

Having a bigger shadow stack is not the only drawback of a Flash-based design. The presence of zero-values and the impossibility to overwrite them, make pop and push operations more expensive. While in a traditional scenario we would always operate on the top of the stack, with FLASHADOW, we might need to interact with any point of the stack. This makes adjourning the shadow stack pointer expensive as we have to find the next address to be popped (highest non-zero value) and the first free entry where to push the new address to (the top of the stack). Considering the ever-increasing

[2] On MSP430, cleared addresses contain `0xFF`.

[3] This allows FLASHADOW to cheaply store values without overwriting

```
int main(void){
  functionA();
  functionA();
  functionA();
}
void functionA(void){
  return;
}
```

Listing 5.1: A simple C program with 3 consecutive calls to an empty function.

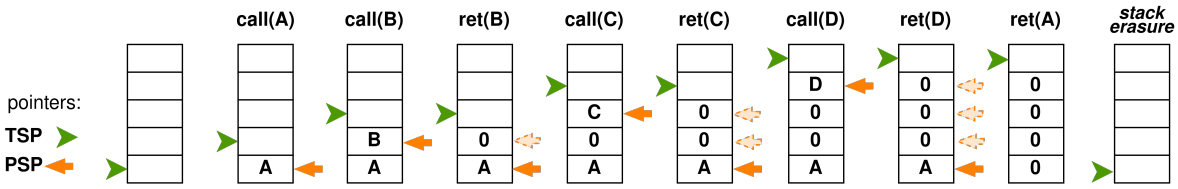


Figure 5.5: Shadow stack representation with the calling sequence `call(A) - call(B) - ret(B) - call(C) - ret(C) - call(D) - ret(D) - ret(A)`. The dashed PSP highlights the travelling. The final stack state represents the stack-erasure optimisation.

size of the stack and its zero-entries, these searches can become expensive in terms of CPU cycles. To overcome this limitation, we propose a two-pointer design with a Top Stack Pointer (TSP) and a Pop Stack Pointer (PSP). The TSP will always point to the top of the shadow stack, i.e. the lowest free slot, thus reducing the cost of push operations. This resembles the behaviour of the normal shadow stack. The PSP is instead used to keep track of the next value to be popped. Keeping this pointer up-to-date is cheap upon a new push (PSP=TSP) but can be quite expensive after a pop operation, considering that the next non-zero value could be at any point of the stack. Figure 5.5 shows an example of FLASHADOW in action. It can be seen how on `ret(c)` and `ret(D)` the PSP has to travel down several positions. Nevertheless, we note that this is the cheaper design that we can realise in a pure-software while being maintained on the Flash.

5.5 Implementation

Although FLASHADOW is compatible with every architecture that supports PISTIS, we propose an implementation of FLASHADOW for the MSP430 architecture from Texas Instruments [71]. MSP430 MCUs are based on a Von Neumann memory model,

and they are widely used in many critical application domains. For instance, they are employed in implantable medical devices (IMDs), e.g. pacemakers, that support standard interfaces for wireless communication [224, 121, 70]. They also have been a target for many research prototypes, including SANCUS [183], SMART [83], and VRASED [185]. We focus on a specific architecture to optimise our implementation and provide a more significant evaluation. Nevertheless, the design could be implemented on a different target architecture, such as Arm Cortex-M.

In particular, we implemented FLASHADOW on top of the latest version of PISTIS, which is implemented on top of the MSP430F5529 MCU that features ∼132 kB of FLASH, ∼8 kB of SRAM, and up to an 8 MHz of CPU speed using internal oscillators. We used HMAC-SHA256 and ChaCha20 from the HACL library [273] to implement other security services, namely $\mathcal{RA}$ and Secure Code Update.

5.5.1 FLASHADOW in numbers

Our FLASHADOW implementation is two-fold: we provide a software-based implementation for the MSP430F5529 MCU and an extension to the GCC compiler, used to produce FLASHADOW-compliant binary images. Both contributions are extensions to the work in PISTIS [103]. Our TCM is a modular software composed of a core, which includes our memory isolation primitive and FLASHADOW, and various TAs that can be deployed independently by adjusting some configurations in a custom Makefile, which serves as an API. The TCM core, including a boot code, Loader/Verifier, virtual functions, shadow stack hooks (pop and push), and some helper functions, is composed of 1232 lines of C code along with 744 lines of Assembly. The cryptographic primitives used, namely HACL HMAC-SHA2 and ChaCha2, comprise 944 lines of code, extracted from the HACL library [273]. Leveraging these primitives, $\mathcal{RA}$ implementation includes 78 lines of code, whereas the secure code update mechanism is composed of 236 lines of C code. The MSP430 architecture features 60 main instructions, of which, 24 instructions had to be instrumented to maintain strong isolation guarantees (further details can be found in [103]). We provide an API, comprising a 121-line Makefile in addition to an extended MSP430F5529 linker script with 601 lines, that fully automates the deployment of the TCM on the target device. Furthermore, our GCC plugin serves as another API to fully automate the production of FLASHADOW-compliant binary images. It comprises a 159-line Makefile that transparently modifies 15 lines of the original linker script of applications and executes 740 lines of python scripts to re-write instructions and embed meta-data. Furthermore, the user API includes a 236-line python script that

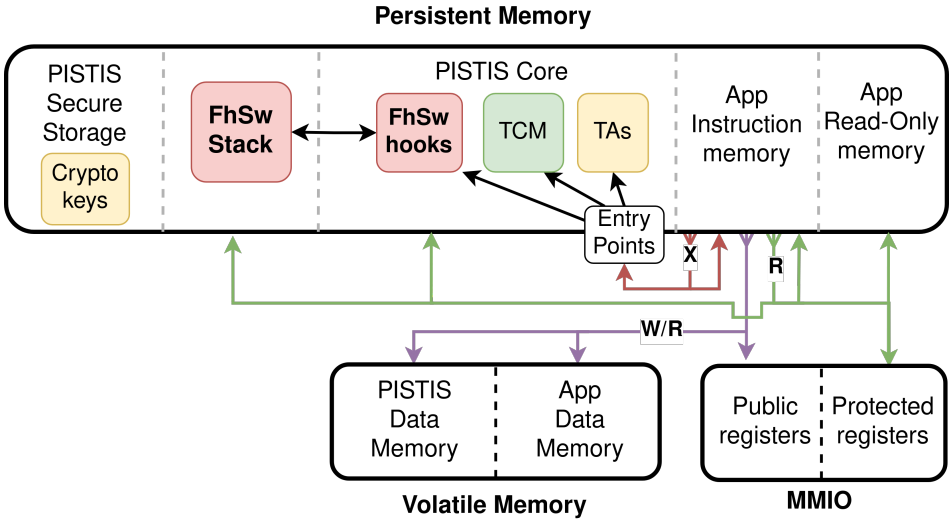


Figure 5.6: FLAShadow (FhSw) memory map on MSP430 architecture.

automates the deployment of produced binaries using serial communication. Notably, these figures include both FLAShadow implementation and the underlying PISTIS.

5.5.2 Memory map in practice

Section 5.3 elaborated on the design rationale of enforcing memory isolation between software modules. As a consequence, Figure 5.3 visualised the resulting memory map w.r.t. the employed access policy (AP). Given that the AP is mainly concerned with read, write, and execute rights, the number of application instructions requiring virtualisation might vary depending on the application's nature.

The implementation of our memory isolation was optimised for our target MSP430 architecture, reducing the number of virtualised instructions by following a slightly different memory map that, nevertheless, has equivalent security guarantees to the one shown in Figure 5.3. The new memory map, visualised in Figure 5.6, introduces a secure storage segment, where all the sensitive data, including cryptographic keys, are stored. This memory part is only accessible by the PISTIS core, relaxing the application access policy as follows: (i) Read access is extended to cover the PISTIS Core, App Instruction memory, PISTIS Data memory and FLAShadow stack; and (ii) Write access is extended to cover PISTIS Data memory.

FLAShadow design rationale is compatible with PISTIS general design, leveraging its virtualisation and verification primitives to achieve a secure shadow stack. However, as section 5.4.2 highlights, we propose a Flash-based shadow stack to meet the implementation proposed in [103]. As such, our FLAShadow implementation leverages

two existing commodity hardware features in MSP430 MCUs: the Bootloader Section (BSL) and the Flash memory controller. Further details follow.

5.5.3 FLASHADOW in practice

Although FLASHADOW was designed as an extension of PISTIS core security primitives, implementing FLASHADOW alongside PISTIS on constrained devices entails many challenges. We recall that PISTIS was designed for the very low-end MCUs, with an implementation that makes a highly optimised use of the few resources available. This leaves little room for additional extensions as the like of FLASHADOW. Nevertheless, we manage to tweak FLASHADOW implementation so to make it compatible with the reference MSP430 target architecture.

Let us consider the two shadow stack registers: TSP and PSP. Although using the available CPU registers makes their operations fast, reserving two additional general-purpose registers (on top of the three already used by PISTIS TCM) might be counterproductive. First, it makes the untrusted application code run slower, having been compiled with one less general-purpose register available. Second, it reduces compatibility with some applications and external libraries.

For instance, instrumenting `libc` with four reserved registers required several tweaks. Instrumenting it with five reserved registers might require considerable effort. Notably, `libc` is heavily used in the MSP430 architecture, due to a limited instruction set where many complex operations are virtualised via stub functions. These auxiliary functions are invoked transparently by the compiler, even without the inclusion of standard `libc` function calls (e.g. `malloc()`).

To tackle this issue, we reduce both TSP and PSP to 10 bit pointers and store both of them in a single 20 bit register: `[TSP«10 | PSP]`. Although merging the two saves one CPU register, it also makes pop and push operations slightly more expensive, requiring additional shift and copy instructions. We reckon that a two-registers FLASHADOW implementation could still be employed for a subset of compatible applications.

We further propose a second optimisation to reduce the cost of return instructions, which might be particularly expensive the more the stack grows. With each pop, we have to traverse the shadow stack to find the highest non-zero value, as seen in Figure 5.5. We call it the *long-travel* side-effect. Although this cannot be avoided in most scenarios, the long-travel becomes unnecessary when our stack only contains zeros, i.e. FLASHADOW is empty. To mitigate this, we propose a stack-erasure optimisation: we erase the entire shadow stack as soon as PSP reaches the bottom, as in the case of

Figure 5.5 after `ret(A)`.

We recall that erasing a Flash segment allows it to be written over once again. On the one hand, this allows both PSP and TSP to restart from the bottom, reducing temporarily the long-travel issue. On the other hand, erasing a segment is an expensive operation costing several milliseconds. Consequently, there is the risk of *over-erasure*, for which applications such as the one in Listing 5.1 would trigger an erasure after each return. This would considerably aggravate the performance of FLAShadow.

To mitigate this issue, and still leverage the performance boost in case of long-travels, we introduce a threshold θ for the TSP, below which the stack is not erased. In other words, we erase the stack only when it is empty and its size has reached θ. It is worth mentioning that θ has an impact on the performance of FLAShadow. An excessively high value would still enable the long-travel issue, while an insufficiently low value would accentuate the over-erasure issue. Notably, the perfect θ is application-dependent, being strictly related to the control flow of an application. Nevertheless, using both statistics and testing we set $\theta = 166$. Algorithm 1 and Algorithm 2 show the pseudo-code of the new virtual functions, `safe_ret` and `safe_call`, respectively.

Limitations

The current FLAShadow implementation has some limitations. First, due to the 10 bit pointers, the shadow stack cannot register more than 511 entries. This poses a hard limit on some recursive applications. In practice, the limit is exacerbated by the ever-growing design which might further reduce the total number of active entries. Indeed, TSP might be higher than 0 even with an empty stack. In this regard, θ has also an effect on the maximum stack size. Lower θ values allow for more frequent stack erasure, thus limiting the aforementioned issue.

5.6 Experimental evaluation

Embedded systems are used with a high variety of low-level applications, differing in the type of operations performed and the hardware resources utilised. This heterogeneity, along with the lack of suitable benchmarks, makes the evaluation of a control flow integrity technique such as FLAShadow non-trivial, requiring a careful selection of a representative set of applications that would reflect the overhead incurred by FLAShadow from various perspectives.

We chose a set of 8 applications, considering three main factors: (i) the compatibility

Algorithm 1: Pseudo code for the assembly `safe_ret` virtual function. `R8` is the reserved register used to store both `PSP` and `TSP`, `SR` is the status register, `SP` is the stack pointer.

```
1  Disable interrupts;
2  Disable FLASH protection;
3  PSP ← fetch from R8 ; /* Mask R8 and add shadow stack base pointer
     */
4  if *(PSP) != *(SP) then
5  |  reset MCU ;        /* Mismatch between stack and shadow stack */
6  end
7  *(PSP) = 0 ;        /* Delete the just popped shadow stack entry */
   /* We prepare the PSP for the next return instruction           */
8  do
9  |  PSP ← PSP-1 ;   /* Decrease PSP until we find a valid entry  */
10 while PSP >= STACK_BOTTOM and *(PSP) == 0;
11 if PSP < STACK_BOTTOM then
12 |  PSP = 0 ;            /* We reached the beginning of the stack */
13 |  TSP ← fetch from R8 ;                          /* Shift R8 */
14 |  if TSP >= θ then /* If threshold is reached then we reset the
   |   stack                                                      */
15 |  |  Erase shadow stack;
16 |  |  TSP = 0;
17 |  |  R8 ← store TSP ;
18 |  end
19 end
20 R8 ← store PSP;
21 Enable FLASH protection;
22 Restore SR;
23 Perform function return;
```

Algorithm 2: Pseudo code for the assembly `safe_call` virtual function. `R6` and `R8` are reserved registers. `R6` is populated with the target function address, `R8` is used to store both `PSP` and `TSP`. `SR` is the status register.

```
1  Disable interrupts;
2  if *(R6) != NOP_SLIDE or *(R6) > APP_TOP or *(R6) < APP_BOTTOM
     then
3  |  reset MCU ;      /* We reset if software tries to break free from
   |   isolation */
4  end
5  Disable FLASH protection;
6  TSP ← fetch from R8 ;                                  /* Shift R8 */
7  if TSP >= MAX_DEPTH then reset MCU;
8  PSP ← TSP ;                        /* We need to match PSP and TSP */
9  *(TSP) = *(SP) ;  /* Save the return address on the shadow stack */
10 TSP ← TSP+1 ;                     /* Increment TSP by one position */
11 R8 ← store both TSP and PSP;
12 Enable FLASH protection;
13 Restore SR;
14 Call R6;                                        /* Make original call */
```

with our target experimental platform (MSP430F5529LP), (ii) the compatibility with FLAShadow limitations, and (iii) making a good balance between the more CPU-intensive and the more IO-intensive tasks that are typical for an embedded device. Our evaluation metrics include memory footprint, execution time, and power consumption.

The proposed FLAShadow implementation is a security primitive built on top of PISTIS TCM [103], bringing new security functionalities at the cost of additional overhead. Notably, FLAShadow brings also several optimisations and improvements to such TCM. Moreover, we use a different compiler version with a different `libc` library, and slightly revised applications to make the tests more representative of our scenario. Finally, the run-time evaluation of FLAShadow was performed with a Saleae Logic Pro 8 logical analyser, rather than with the internal clock timers of the reference board. As a result, the baseline data for FLAShadow might differ from the data in the tables of PISTIS [103].

NOTE. Unless otherwise specified, applications are compiled using −O3 optimisation flag and 8 MHz as a CPU speed.

Table 5.1: Memory footprint and execution time comparison between original applications (Orig.) and FLASHADOW-enabled ones (Mod.).

App	Memory Footprint		Execution Time	
	Orig.	Mod.	Orig.	Mod.
SerialMSP	296 B	346 B (+**16.89%**)	274.06 ms	275.44 ms (**0.50%**)
CopyDMA	452 B	544 B (+**20.35%**)	118.90 ms	806.33 ms (+**578.16%**)
Bitcount	1432 B	1696 B (+**18.44%**)	5.4609 ms	5.7412 ms (+**5.13%**)
ML-acc	5422 B	8612 B (+**58.83%**)	3985.4 ms	16407 ms (+**311.69%**)
16bitSwitch	134 B	148 B (+**10.45%**)	0.00759 ms	0.00760 ms (**0.13%**)
8bitMatrix	876 B	878 B (+**0.23%**)	0.5498 ms	0.5499 ms (**0.02%**)
MatrixMul	524 B	526 B (+**0.38%**)	0.3280 ms	0.3290 ms (**0.30%**)
dhrystone	1270 B	2308 B (+**81.73%**)	100.42 ms	386.25 ms (+**284.63%**)
Average		+**25.91%**		+**168.58%**

5.6.1 Memory footprint

Needless to say that instrumentation adds extra bytes to the size of each application due to inserting extra instructions, e.g. NOP slides, virtual calls, etc. Considering a GCC toolchain with a −O3 optimisation flag (that further optimises the execution time), Table 5.1 shows the size differences (in bytes) between binaries compiled using a standard GCC toolchain, and our custom toolchain that adds the required instrumentation. In other words, the recorded sizes represent the amount of consumed Flash memory in bytes.

Although FLASHADOW adds some instrumentation on top of PISTIS, e.g. requiring all calls to be instrumented, it also makes some instrumentation redundant (e.g. return NOP slides). Furthermore, FLASHADOW brings several optimisations to the TCM, ultimately decreasing the memory overhead to an average of 25.91% (from the 33.74% of native PISTIS [103]).

Table 5.1 shows the comparison of the binaries with FLASHADOW. Notably, the results are application-dependent, with the memory footprint increasing depending on the types of instructions used. Furthermore, none of the tested applications incurred issues due to the increased memory size. We believe that in practical scenarios, this memory increase can be acceptable and it is compatible with the memory capacity of class I devices [125]. Interestingly, by inspecting the instrumented binaries, it can be noticed how in some cases, a considerable part of the additional memory footprint lies within the `libc` stub functions.

As shown in Figure 5.6, both PISTIS TCM and FLASHADOW persistently occupy part of the Flash memory. Compared to the original size of the native PISTIS core, which is 7.4 kB, FLASHADOW brings some optimisations that result in a reduced core of 7.2 kB, corresponding to the 5.5% of the available Flash memory(∼132 kB) of the target

MCU. Although FLAShadow expands the multi-fold functionality of PISTIS, it also makes several optimisations to the core of it, leading to a smaller memory footprint.

5.6.2 Execution time

Similar to all control-flow integrity techniques, FLAShadow imposes a run-time overhead on the applications' execution times, due to the need to perform run-time checks on their control transfer instructions. In order to evaluate the impact of FLAShadow on the normal execution of applications, we measured the difference between the time it takes to execute our test applications with and without FLAShadow. Measurements are recorded at a CPU speed of 1 MHz, considering an average of 5 different test runs for each one. Table 5.1 shows the overhead imposed by FLAShadow, which increases the execution time by an average of 168.58%.

As would be expected, FLAShadow overhead is greater than the one of the native PISTIS (40.30%) due to the need of inserting the shadow stack operations on all calls and return statements. The resulting overhead averages at 168.58%, highlighting how the cost of a software-only shadow stack can be prohibitive in several scenarios. However, it can be observed how the results are highly application-dependent, with some that incur an overhead of around 5%. In particular, the overhead is high in scenarios with an increased number of function calls, while being negligible in other cases.

It is important to mention that this might be an acceptable price to pay for accommodating a full-featured Trusted Computing architecture with an integrated shadow stack without any hardware modification, especially in those scenarios where security is critical. In such cases, the performance drop might be overshadowed by the increased security, thus allowing the safe execution of the untrusted application. We also note that using other optimisation flags rather than `-O3` might optimise the execution time for some applications. Finally, we note that the accompanied $\mathcal{RA}$ with PISTIS consumes 7.4 seconds when computing a message authentication code (MAC) digest over a 64 kB memory block using the default clock speed (8 MHz), considering the HACL HMAC-SHA2 cryptographic primitive [273].

To further analyse the results, we performed a deep inspection of the instrumented code — and of its execution — of each application. From this analysis, we could determine that a big part of the overhead can be ascribed to the high number of function calls included by the MSP430 `libc` library, which is transparently used by the compiler, comprising many stub functions that are crucial to the MSP430 architecture. However,

these functions make heavy use of the shadow stack. We reckon that a standard library crafted explicitly for our use case, and thus with a reduced number of function calls, could boost the performance of FLASHADOW. Similarly, it is possible to decrease the application overhead by performing a code-in-lining operation on the application code. This would produce denser yet faster code, thus increasing the compatibility of FLASHADOW with the more performance-demanding applications.

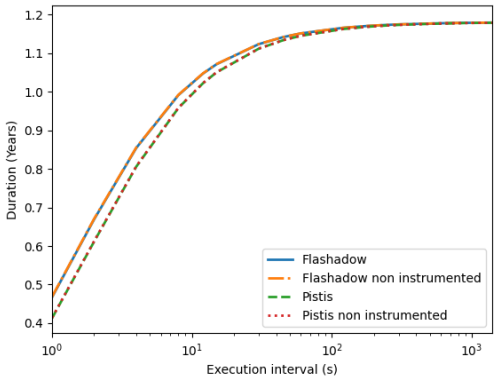
5.6.3 Power consumption

Embedded devices are often used in areas where a connection to a power line is simply unfeasible or expensive. Therefore, such devices depend on batteries as their main power source, requiring optimised energy usage to reduce maintenance costs.

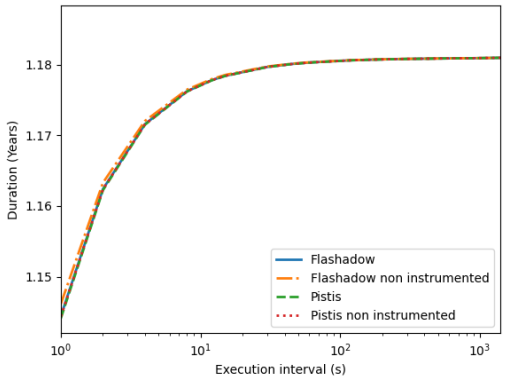
Figure 5.7 compares the amount of consumed energy by applications compiled with and without FLASHADOW. Three different types of applications are considered: I/O-, memory-, and CPU-intensive. Considering a 3.0 V / 3 Ah battery, Figure 5.7 shows the impact on battery life when considering different execution rates. In the worst-case scenario, our exemplar applications, namely SerialMSP, Bitcount and CopyDMA, decrease the battery lifetime by -0.30%, -0.15% and -69.69% respectively, when running once a second. The battery lifetime degradation becomes lower than 0,01% when running the aforementioned applications at rate of once every 120 m, 30 m and 360 h, respectively.

5.6.4 FLASHADOW in real-world scenarios

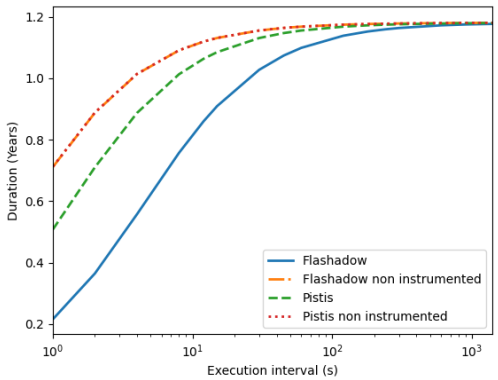
The security protection introduced by FLASHADOW comes with a memory and performance price that might not always be acceptable. In particular, the high run-time overhead might be prohibitive in real-time applications where FLASHADOW could break their functionality, or in scenarios where performance is more important than security. However, we argue that many real-world applications deployed on these very low-constrained devices can afford the performance drop to benefit from the protection of FLASHADOW. Let us consider a few real-world examples: a smart lock [230], a health monitor system [122] and an agriculture irrigation node [147]. In such instances, the embedded applications might have an execution time in the order of fractions of a second, which becomes almost irrelevant to the functionality of the device. In particular, it is of uttermost importance for these devices to correctly carry out their task, regardless of whether this is accomplished with a delay. Consequently, FLASHADOW run-time overhead would not disrupt these applications' functionality, but its protection would fill their security gap and prevent potentially catastrophic attacks.



(a) SerialMSP



(b) Bitcount



(c) CopyDMA

Figure 5.7: An analysis of the power consumption of three exemplar applications when running with their original binary, with a native PISTIS-compliant binary and with a FLASHADOW-compliant binary.

Furthermore, we reckon that in practice, the overhead introduced by FLASHADOW in these real-world scenarios can be much smaller. These embedded applications are seldom computationally intensive, focusing rather on some interaction with the outside world: examples are sensors, actuators or even network nodes. In such cases, the process of data acquisition or transmission introduces delays that make most computational overhead negligible. We demonstrate one such case by emulating a real-world case on an MSP430-based device equipped with a gyroscope sensor and a low-power IEEE 802.15.4 compliant radio. The main task of the device was to obtain 10 different measurements from the gyroscope (x,y,z coordinates), encrypt them, transmit the encrypted data over the network to a host controller, and wait for an acknowledgement. By measuring the time required to perform this entire set of actions (that constitutes the job of the IoT device), we noticed that out of a total of 276 ms as a complete

execution time, only **6.02%** of the time was occupied by the software instrumented by FLASHADOW. The majority of the time (93.98%) was consumed by the network layer (that is hosted in another MCU and interfaced with through peripherals). This means that if FLASHADOW incurs 168% performance overhead on top of the normal execution time of the corresponding software module, the overall performance overhead on top of the total execution time of the entire job will be not more than 10.1%. Therefore, we believe that the run-time overhead incurred by FLASHADOW can be reasonable in practice.

In conclusion, FLASHADOW can be a valid and practical security solution in many security sensitive contexts where more sophisticated security hardware is too expensive or impractical to be deployed, and software-based techniques are the only viable option.

5.6.5 Security evaluation

FLASHADOW is designed with the goal of thwarting ROP attacks, thus increasing the security of vulnerable applications against control-flow attacks. Considering the attacker model described in Section 5.3.1, we proceed to demonstrate the effectiveness of FLASHADOW. Let us consider an application with a buffer overflow that grants the attacker the ability to write to any arbitrary memory address.

First, we consider the case in which the attacker tries to directly modify a return address on the stack. In such a case, even a successful corruption would not alter the application control flow since FLASHADOW fetches its return addresses from the separate shadow stack. Second, the attacker could try to directly modify the values on the shadow stack instead, by either attempting a direct injection or by hijacking the virtual function `safe_call`. Both these attacks would fail due to the intrinsic write protection granted by PISTIS TCM, which prevents any unauthorised write on Flash, where both the shadow stack and the code of the virtual functions are stored.

A more sophisticated attacker could try to circumvent the use of the shadow stack by crafting code that does not use the `safe_return` virtual call. This could be achieved by either tampering with the compiled binary or by tampering with the toolchain itself. However, the Loader/Verifier in the TCM thwarts such an attack by ensuring that the executed binary does not contain unprotected return statements and that those encoded in variable-length instructions are inaccessible (Section 5.3.2). Finally, a different attack vector for ROP is the use of exceptions by trying to corrupt the return value at the end of the exception frame. However, PISTIS intrinsically protects the stack frame return value by storing it in the write-protected Flash and then using `safe_reti` to fetch it.

In order to further show the effectiveness of FLAShadow, we created a demo application vulnerable to ROP attacks, and then tested how FLAShadow can protect it. We chose to simulate a smart lock service, as it is a common security-critical IoT use cases that has been proven to have vulnerabilities.[4] In particular, our smart lock (i) receives a pin from the user, (ii) compares it with a pre-defined correct pin and (iii) opens the lock if the two pins match. For the sake of simplicity, we hard-coded the insertion of the pin in the application code, bypassing a more realistic interaction between the user and a peripheral (e.g. a keyboard). Moreover, rather than writing to some MMIO registers to open an actual lock, we simply turn on a red LED to simulate an open lock. These two simplifications are reasonable since they do not alter the logic of the application.

One of the prerequisites for an ROP attack is a memory vulnerability, which can be very common in C and C++ programs. As such, we introduced a buffer overflow in the `receivePin` function, thus allowing potential attackers to corrupt the stack. In practical and more complex ROP attacks, the attacker could leverage this vulnerability to chain several ROP gadgets, as in [3], and execute Turing-complete code. We start by simply demonstrating the feasibility of ROP attacks by simulating a malicious over-length user input that corrupts the stack, overwriting the return address of the `receivePin` function with the address of the `openLock` function. We tested this attack in an unprotected environment, where it manages to divert the control flow of the application to an illegal path, successfully triggering the opening of the lock. Next, we tested a more sophisticated ROP attack, in which the attacker chains multiple ROP gadgets to unlock the smart lock. Our analysis of the application did not reveal any suitable ROP gadgets, so we manually introduced two, as shown in Listing 5.2. We then corrupted the stack with the addresses of the first gadget, the `openLock` function, and the second gadget. The attack was successful: (i) the application returns from the `receivePin` function to the first gadget, (ii) the first gadget retrieves a value (the address of `oepnLock()`) from the corrupted stack and stores it in R12, (iii) execution then returns to the second gadget, which (iv) calls the value stored in R12, redirecting control flow to `openLock()` and successfully unlocking the device.

To demonstrate the effectiveness of our shadow stack, we tested the same application and the same attacks on our MSP430 device with FLAShadow installed. When our

[4]Some examples include CVE-2022-32504, CVE-2022-32502, and CVE-2019-17518. A more comprehensive study could explore the prevalence of these vulnerabilities in embedded systems, by analyzing firmware from various vendors and architectures to assess its susceptibility to ROP and other run-time attacks that FLAShadow would mitigate.

```
//ROP Gadget 1
MOV @SP+, R12      //Pop a value from the stack to R12
RET                //Return

//ROP Gadget 2
CALL R12           //Dynamic call to an address in R12
RET                //Return
```

Listing 5.2: Two ROP gadgets inserted in our Smart Lock application.

shadow stack is in place, the attacker still manages to overwrite the return address on the stack (either with the address of the `openLock` function or the address of one of our ROP gadgets). However, in both cases FLASHADOW forces the MCU to trigger a reset as soon as the `receivePin` function tries to return to the corrupted address. These simple Proofs of Concept (POCs) thus demonstrate how FLASHADOW can effectively protect the backward edges of the application, effectively defending it from ROP attacks.

5.7 Discussion

5.7.1 FLASHADOW on other architectures

We developed FLASHADOW on the MSP430 architecture due to its widespread use in both research and industry. Additionally, its constrained nature provides an excellent testbed for demonstrating the feasibility of advanced software-based security services. However, FLASHADOW is not strictly tied to this architecture. While Section 5.8 discussed solutions available for more feature-rich architectures — such as Silhouette [270] for ARMv7-m — a wide range of architectures still lack the support for hardware-based memory protection, e.g. an Memory Protection Unit (MPU). Examples include AVR, ARMv6-M (found in Cortex-M0), and certain RISC-V implementations without an Physical Memory Protection (PMP).

We believe FLASHADOW could be ported to such architectures by extending its TCM support accordingly. The only requirements are a memory isolation primitive and virtualized functions to replace return and call instructions. Notably, Flash memory is a standard feature in most embedded devices, typically subject to the same write constraints described in Section 5.5. This consistency simplifies the porting process for FLASHADOW across different platforms.

5.7.2 Forward-edge protection

FLAShadow is a CFI scheme focused on backward-edge protection for untrusted applications. It safeguards the integrity of return addresses on the stack, ensuring that each function returns to its legitimate caller. While this effectively prevents ROP attacks, it does not protect against other control-flow attacks, such as JOP, which target forward edges.

Although the TCM already provides partial forward-edge protection, as discussed in Section 5.3.2, it could be further refined to obtain a fine-grained protection. Currently, the TCM always checks if the destination of dynamic jumps is valid. One possible enhancement is extending the TCM to maintain a table of valid function pointers, allowing the TCM's virtual functions to also verify if the jump destination belongs to the correct flow. This approach would require analyzing the application's Control-Flow Graph (CFG) to extract valid function pointers.

For example, consider the dynamic call in Listing 5.2, `CALL R12`. This instruction enables function invocation via a register-stored address. Instead of merely verifying that the register holds a valid address (as done using the NOP slide mechanism described in Figure 5.4), we could check the address against a predefined table of valid function pointers for that specific callee.

This enhanced protection would likely introduce additional memory overhead, as the table of valid function pointers must be stored in the TCM. Additionally, it would impose a performance cost, as each dynamic call would require a table lookup.

5.8 Related work

The research area of control-flow hijacking attacks and defences has been active for more than two decades with various proposals for detecting or protecting against attacks. Nevertheless, many of the proposed defences are not compatible with low-end embedded systems, whose protection is often hindered by the lack of proper security hardware. Therefore, the focus of this section will be on discussing related work that targets MCUs, which represent the core of any embedded device. That being said, Control Flow Integrity (CFI) and Control Flow Attestation (CFA) are the most common defence approaches that have been proposed for embedded systems.

Control-Flow Integrity (CFI). CFI is a security technique that is used to mitigate control-flow hijacking attacks by restricting the set of possible control-flow transfers to

those that are strictly required for correct program execution. In general, CFI solutions instrument inline reference monitors for Indirect Control Transfer (ICT) instructions, whose transfer targets could be compromised, to enforce that ICT instructions only jump to legitimate targets at run-time. The CFI enforcement policy is enforced according to a legitimate Control-Flow Graph (CFG) that is generated during compile-time. Notable is that there are various variations of CFI mechanisms enforcing different levels of CFGs to achieve different degrees of attack surface reduction. In other words, there is a wide range of CFI implementations, each makes its own choice of CFI policy, i.e. what precision of CFGs it enforces.

CFI-CaRE [188] proposes a forward-edge CFI mechanism along with a shadow stack targeting ARMv8-M-based MCUs to leverage TrustZone-M [23] for hardware isolation and protection of metadata. SCFP [264] extends a RISC-V core designed for embedded devices with a hardware module between the CPU's fetch and decode stage, which continuously authenticates instructions, yielding a fine-grained control-flow integrity scheme for both forward and backward edges. Similar to CFI-CaRE, RECFISH [260] also proposes a forward-edge CFI mechanism with a shadow stack targeting closed-source binaries, running on top of Arm Cortex-R MCUs with real-time operating systems. It uses binary instrumentation and Memory Protection Unit (MPU) to place the shadow stack in a privileged region, requiring a system call with each return. Notable is that both CFI-CaRE and RECFISH incur a high-performance overhead in many cases, i.e. $\approx 500\%$ for some applications. μRAI [11] mitigates ROP attacks by replacing all return instructions with hard-coded jump tables, where the address of the right return entry is encoded in a single reserved register that is never spilt into memory without protection. To handle cases such as calling non-instrumented libraries and interrupt routines that execute in a higher privilege level, μRAI depends on a variant of the Software Fault Isolation (SFI) approach [259] along with the existence of an MPU. Silhouette [270] targets also ROP attacks by proposing an MPU-protected shadow stack for Arm-based MCUs. It creates the logical separation between code and memory by utilising the possibility of executing store instructions in either privileged or unprivileged mode. The proposed shadow stack can only be modified using privileged store instructions. The MPU is used to enforce memory access rules. FH-CFI [90] utilizes a hash-based message authentication code scheme to encrypt and decrypt return and call-site instructions to defeat control-flow hijacking attacks. It mainly targets Arm MCUs, and thus depends on some hardware features to protect encryption keys. Kage [77] is a software system that mitigates control-flow hijacking attacks on MCUs with real-time embedded operating systems. It consists of a kernel extension that enforces protection

at run-time by leveraging an MPU, and a compiler that transforms code to be Kage-compliant. Kage can be seen as an enhancement of Silhouette [270] in terms of enabling intra-address space isolation, separating control and non-control data in two different parts of memory. de2019fixer [72] is a hardware extension for embedded RISC-V processors to guarantee control flow integrity against both JOP and ROP attacks.

All of the aforementioned approaches offer different performance and, more importantly, provide different security guarantees. We compare the latter in Table 5.2. Notably, it can be seen how FLAShadow is the only system that effectively supports the use of an untrusted toolchain while also protecting the interrupts routines. Specifically, the use of an untrusted toolchain guards the device against the tampering of the application binaries or of the toolchain itself. The interrupts protection prevents any vulnerabilities in the code used to handle interrupts and exceptions from being exploited by an attacker to mount ROP attacks. Both of these security guarantees considerably lower the attack surface, guarding against stronger attacker models, as shown in Section 5.6.5. Regardless of the pros and cons of the aforementioned schemes, they all depend on some sort of hardware assistance to enforce protection. As Table 5.2 highlights, this assistance varies from the introduction of custom hardware to the use of specific security components. Consequently, these solutions become impractical and expensive, requiring the replacement and redeployment of a considerable number of legacy, cheap and very low-end devices. On the contrary, FLAShadow is a pure-software approach targeting embedded systems with zero hardware security features, thus filling this security gap by allowing the re-use of existing devices without compromising the security of the system. As argued in Section 5.6.4, this can be most needed in security sensitive scenarios.

Control-Flow Attestation (CFA). CFA has emerged as a technique for attesting the integrity of the execution state of applications, following the main principles of static attestation mechanisms [16]. The main argument behind proposing CFA was that naively integrating CFI approaches into static remote attestation protocols would provide limited state information to the remote verifier. In particular, CFI techniques can only report whether a control-flow attack occurred and provide no information about the actually executed control-flow path. Therefore, the verifier cannot determine which path has been hijacked to divert the control-flow execution.

C-FLAT [4] is the first CFA proposal for embedded systems. It assigns a unique ID for each basic block of code, and instruments control-transfer instructions to log the IDs of their blocks in a protected memory area, targeting and leveraging TrustZone-

Table 5.2: FLASHADOW vs. state-of-the-art backward CFI defences from various perspectives.

CFI Defence	HW Requirements	Untrusted Toolchain	Interrupts Routines Protection
de2019fixer [72]	Custom HW ✗	Yes ✓	No ✗
CFI-CaRE [188]	TrustZone + MPU ✗	No ✗	Yes ✓
SCFP [264]	Custom HW ✗	No ✗	Yes ✓
RECFISH [260]	MPU ✗	No ✗	No ✗
μRAI [11]	MPU ✗	No ✗	Yes ✓
Silhouette [270]	MPU ✗	No ✗	No ✗
Kage [77]	MPU ✗	No ✗	Yes ✓
FLASHADOW	None ✓	Yes ✓	Yes ✓

enabled MCUs. A hash chain of the accumulated logs will be calculated and sent to the verifier upon request. The verifier would then verify the received report based on some reference measurements that are calculated depending on a pre-generated Control-Flow Graph (CFG) during compile-time. Lo-FAT [75] overcomes the high-performance overhead issue of C-Flat (due to software instrumentation) by proposing a pure-hardware CFA scheme, extending a RISC-V core with hardware monitors. ATRIUM [267] extends LO-FAT by considering a stronger adversary that is capable of mounting some sort of physical attack. Tiny-CFA [186] is a software-hardware co-design for a CFA scheme that requires minimal hardware modifications, and thus cheaper than LO-FAT and ATRIUM. To the best of our knowledge, all existing CFA schemes are hardware-assisted. Therefore, they are infeasible for simple embedded systems that are already manufactured and cannot be modified. Furthermore, compared to FLASHADOW, while such schemes are able to detect ROP attacks, they do not provide any prevention capabilities. In critical systems where detection is not enough, FLASHADOW can provide stronger security guarantees to prevent ROP attacks at run-time.

5.9 Summary

In summary, this chapter proposed FLASHADOW, a software-based shadow stack design to mitigate ROP attacks on low-end embedded systems. We designed and implemented FLASHADOW as an extension of PISTIS, our software-based TEE. The key idea was to provide an integrity-protected memory area on the Flash leveraging software-based memory isolation, where a shadow stack can be securely maintained.

By evaluating our implementation both with benchmarks and with real-life applications, we strived to answer the question of how feasible it is to protect the lowest-end devices from ROP attacks. We showed how a solution such as FLASHADOW might

be reasonable in some real-world scenarios, offering strong security guarantees with a practical overhead. Considering that the state-of-the-art does not offer comprehensive solutions to these devices due to the lack of security hardware, FLAShadow can be seen as an important step towards securing these devices. Such a solution fills a security gap for which other existing solutions might be too expensive or unpractical. Interestingly, our design could be further extended to introduce a fine-grained control flow integrity scheme for forward edges that can be seamlessly integrated with FLAShadow.

Chapter 6

TAGShield: Persistent Tagging for Comprehensive Stack Memory Error Mitigation

This chapter is based on a currently *work in progress* paper:

Grisafi, M., Ramponi, C., Ammar, M., Crispo, B. (2025). TAGShield: Persistent Tagging for Robust Stack Memory Error Protection.

Preamble

In the previous chapters, we have explored a comprehensive software-based security stack appositely designed to address the lack of hardware-based Trusted Computing solutions on many low-end systems. While software-only approaches offer advantages such as higher scalability and cross-compatibility, more hardware-dependent techniques often remain the preferred choice for devices that support them, primarily due to performance considerations. Security-related hardware features are gaining traction primarily in high-end devices due to their cost and complexity. While efforts have been made to extend these solutions to embedded systems — such as TrustZone-M for ARMv8-M Cortex-M devices — this trend remains inconsistent. For instance, the recent Cortex-R architecture, which is designed for real-time and safety-critical systems, lacks any TEE technology. On the other hand, traditional high-end devices, such as those equipped with ARMv9 chipsets, are being launched with even more sophisticated security features, such as the Realm Management Extension (RME) which adds an extra secure

enclave on top of the TrustZone Secure World. Although these solutions are purely hardware-based, they still require a software layer to properly manage them. Software-hardware co-designs are therefore pivotal on high-end systems as well, but, contrarily to low-end devices, they are more hardware centred. In this chapter we shift the focus from low-end devices to high-end devices, exploring the role of software-hardware co-designs in the presence of advanced hardware security features.

Interestingly, Trusted Computing is not the only area being explored by chip manufacturers. Beyond TPMs and TEEs, manufacturers are advancing the development of innovative hardware components to provide enhanced, performance-friendly security guarantees, such as pointer integrity. Two notable examples, recently introduced by Arm, are the Memory Tagging Extension (MTE) and Pointer Authentication Code (PAC). These features can strengthen vulnerable applications against common attack vectors, such as buffer overflows, by safeguarding their memory.[1] Although these solutions are hardware-based, they still rely on carefully designed software instrumentation to enable and manage them effectively. Notably, the security guarantees of these hardware modules are entirely dependent on the software that administers them, making for an excellent case study on software-hardware co-designs.

In this chapter, we focus on Arm MTE and explore how software can go beyond the basic configuration of this module to address its limitations. Specifically, we introduce TAGShield, an enhanced memory tagging schema that serves as a run-time exploit mitigation mechanism that leverages MTE to provide a persistent and deterministic defense against stack-based spatial memory errors, while preserving the integrity of pointer tags. TAGShield provides strong security guarantees through a transparent compile-time tagging scheme, complemented by a lightweight software instrumentation layer that enforces tag integrity. Evaluation results demonstrate that TAGShield introduces a geometric mean runtime overhead of less than 25% on representative benchmarks, including SPEC CPU 2006 and nginx, while effectively mitigating all stack-based spatial memory vulnerabilities in the Juliet C/C++ test suite. Notably, Arm MTE is available only on select high-end devices with ARMv8.5 or newer chipsets, setting TAGShield apart from the contributions of previous chapters. Rather than targeting lower-end devices—where software-hardware co-designs rely heavily on software—TAGShield introduces only a thin software layer atop the existing hardware. This layer efficiently manages Arm memory tagging to ensure effective and robust memory protection.

[1] Compared to Trusted Computing, which aims at providing a secure execution environment for security services, these techniques are designed to harden the single applications against run-time attacks.

Although TAGShield is software-hardware co-design that goes beyond the scope of Trusted Computing, we explore how this solution can help strengthen Trusted Applications, effectively boosting the security of Trusted Computing.

Chapter outline. This chapter is organised as follows. Section 6.1 provides an overview of our approach while Section 6.2 motivates the need for TAGShield and provides some background on Arm MTE and stack spatial memory errors. Section 6.3 and Section 6.4 will describe the design and implementation of our solution, respectively, while Section 6.5 will evaluate its impact on performance by analysing two benchmark suites and three real-world applications. Section 6.6 will provide a security analysis, go over future work and discuss alternative designs. Section 6.7 will provide an in-depth comparison with other MTE-based security solutions, thus showing how our software layer can strengthen this technology. Finally, Section 6.8 concludes this chapter.

6.1 Introduction

Memory corruption vulnerabilities, particularly spatial memory errors, continue to dominate the landscape of modern software exploits. According to recent findings from Google [98], spatial memory errors account for over 40% of memory corruption vulnerabilities reported in 2024, highlighting the critical need for robust defence mechanisms. These errors occur when a program accesses memory beyond the intended boundaries, enabling adversaries to exploit the resulting vulnerabilities to manipulate program execution and launch sophisticated exploits, such as control-flow hijacking and data-only attacks. Stack-based memory errors, in particular, are prime targets for such attacks, and their exploitation remains an ongoing challenge [118, 120, 92].

Despite the longstanding recognition of stack-based spatial memory errors as a critical threat, first emphasised in the Anderson report [18], these vulnerabilities remain a pressing challenge in modern software security [98]. While defences have evolved to safeguard return addresses — historically using stack canaries [64] and now employing hardware-enforced shadow stacks [132] — these measures only address a subset of stack-specific spatial memory errors. They fail to mitigate attacks that exploit modifications to stack-allocated local variables or unauthorised writes or reads of stack memory, which can lead to sophisticated exploits such as data-only attacks [56]. Probabilistic defences, such as ASLR [38], add complexity to an adversary's efforts but are often bypassed by exploiting relative offsets on the stack to leak certain addresses [207].

Researchers have tackled the challenge of stack-based spatial memory errors by initially exploring comprehensive memory safety defences designed to address all types of memory corruption [36, 79, 193]. However, these approaches often introduce significant performance overheads, frequently exceeding 2x, which makes them impractical for many real-world applications. To mitigate these challenges, researchers have classified memory errors into three primary categories: spatial errors, which occur when a program accesses memory outside its intended bounds; type errors, which arise from incorrect interpretations of memory contents; and temporal errors, which involve the use of deallocated or invalid memory. Techniques specifically targeting spatial errors have emerged in several forms. Isolation-based mechanisms, such as Safe Stack [256] and DataGuard [118], attempt to isolate safe objects from unsafe ones, but they cannot prevent attacks that occur between unsafe objects. Randomisation-based defences, like SaViouR [151], randomise the stack layout to make exploitation harder, but they suffer from compatibility issues and offer limited security guarantees. Binary rewriting techniques, such as StackArmor [53], rewrite binaries to enforce safety properties but are often hindered by high run-time overhead and the risk of false positives. Software-based bounds-checking mechanisms, such as SoftBounds [179], Buggy Bounds [8], and Low Fat Pointers [80], introduce pointer- or object-based bounds checking but face drawbacks such as substantial performance costs and limited compatibility.

To balance security and performance, hardware-assisted memory safety features in modern architectures offer a promising solution to mitigate memory corruption vulnerabilities. Among these approaches, Arm's Memory Tagging Extensions (MTE) emerges as a significant advancement in addressing spatial memory errors [25]. MTE associates every memory location and pointer with a 4-bit tag, and enforce a hardware check to ensure that pointer dereferences only occur when the pointer and memory tags match. While MTE-based solutions [110, 54, 203, 101, 217] reduce the run-time overhead compared to software-only approaches [179, 8, 80, 143], they come with significant limitations. Most defenses leveraging MTE rely on hardware-generated random tags, which offer only probabilistic security guarantees [110, 54, 203]. The low entropy of these tags exposes applications to brute-force attacks, while tag collisions between neighboring memory objects leave systems vulnerable to contiguous overflows and type confusion exploits.

StickyTags [101] addresses these limitations by adopting a different approach, which leverages persistent memory tagging. By organizing the stack and heap layout into per-size-class regions, StickyTags applies persistent memory tags to each region in a predetermined pattern, eliminating the need for costly memory re-tagging. This de-

sign enables low-cost deterministic protection against stack memory errors. Despite its small performance footprint, StickyTags shares a fundamental limitation inherent to all current MTE-based approaches. These solutions are primarily tailored for *sanitisation* rather than run-time exploit mitigation, even though some attempt to address the latter [101, 217]. Crucially, *they do not guarantee the integrity of pointer tags*, leaving the unused upper bits of pointers - of which part is reserved for tags - vulnerable to manipulation by malicious pointer arithmetic operations. This limitation poses a significant barrier to the adoption of the aforementioned solutions, as the gap remains substantial. The sustained efforts of major companies, exemplified by Apple's 2023 `-fbounds-safety` LLVM proposal [177] and Google's 2024 STL hardening solution [99], highlight the importance of this problem and continues to demand attention.

Contribution. In this paper, we present TAGShield, a run-time exploit mitigation mechanism designed to defend against stack-based spatial memory errors. TAGShield combines memory tagging with selective software instrumentation to address the limitations of prior techniques, providing deterministic protection while preserving the integrity of pointer tags. This approach effectively mitigates all pointer corruption attacks, including malicious arithmetic manipulations, providing a robust defense against stack-based spatial memory errors.

The core of TAGShield's design is based on two key principles:

- Transparent Memory Coloring Scheme: TAGShield employs an efficient memory coloring mechanism to selectively tag unsafe pointers and objects, ensuring spatial isolation and preventing memory collisions between adjacent regions.
- Lightweight Software Instrumentation: TAGShield introduces a software layer that persistently enforces the integrity of pointer tags, making them immutable and resistant to corruption. This layer ensures that adversarial attempts to manipulate tags are thwarted.

TAGShield strikes an optimal balance between performance and security through a selective protection strategy that focuses on vulnerable stack allocations. It also leverages compile-time tag generation to significantly reduce run-time overhead while maintaining robust security guarantees. Notably, TAGShield outperforms all state-of-the-art bounds-checking mechanisms with comparable security guarantees. Furthermore, its design introduces no compatibility issues, minimizing barriers to adoption in real-world applications.

6.2 Background and motivation

6.2.1 Stack spatial memory errors

A spatial memory error occurs when a program accesses memory outside the bounds of the intended referent, violating memory safety principles. Such errors are prevalent in low-level programming languages like C and C++, where developers have direct control over memory management. The intended referent of a pointer is the object from which the pointer's base address was derived, as defined in [29]. When a pointer accesses memory outside this defined boundary, it can lead to unauthorised reads or writes, commonly known as buffer over-read and buffer overflow, respectively.

Spatial memory errors have been a well-documented issue in computer science for over five decades [18]. Despite this long-standing awareness, these errors remain a significant challenge to detect and mitigate effectively. Automated static analyses often fail to catch subtle spatial violations due to the complex control flow and data structures in modern software systems [49]. Similarly, dynamic analysis techniques such as fuzzing may only uncover spatial memory errors that occur in specific execution paths, leaving many vulnerabilities latent in the codebase [155]. A notable subset of spatial memory errors involves stack-based buffer overflows, where a program writes more data to a stack buffer than it can hold, corrupting adjacent memory. These errors can be particularly dangerous, as they often lead to control- and data-flow hijacking attacks, where an attacker manipulates the corrupted stack to execute arbitrary code or control the program's behaviour. Out-of-bounds writes, which mainly include stack buffer overflows, rank second in the CWE Top 25 for 2024, following cross-site scripting (XSS) and preceding SQL injection [67]. This list, based on high-severity vulnerabilities reported between mid-2023 and mid-2024, highlights the critical impact of buffer overflow vulnerabilities in modern software security. For instance, as reported in [92], security researchers have identified over 200 stack buffer overflow vulnerabilities in the past three years, with 122 of these classified as high-severity, scoring 7 or above on the CVSS[2] scale.

Recent examples of high-impact stack buffer overflow vulnerabilities further highlight the pervasiveness of spatial memory errors across different software domains. For instance, CVE-2023-21610 affected Adobe Acrobat Reader, allowing an attacker to execute arbitrary code through a specially crafted PDF file, leading to a stack-based buffer overflow. Similarly, CVE-2023-22243 impacted Adobe Animate, where a stack buffer

[2]Common Vulnerability Scoring System.

overflow could be exploited to achieve remote code execution. In the realm of cryptographic software, CVE-2022-3786 was a vulnerability in OpenSSL, one of the most widely used libraries for secure communication. This flaw allowed a buffer overflow in the X.509 certificate verification process, which could be exploited by attackers to cause a denial of service or execute arbitrary code. In the Linux kernel, CVE-2024-1151 exposed a vulnerability in the Open vSwitch (OVS) component, where recursive operations could lead to a stack overflow, potentially causing system crashes or other denial-of-service conditions. Another recent issue, CVE-2024-24861, was identified in the Xceive XC4000 silicon tuner device driver, where an integer overflow could trigger a stack-based buffer overflow, leading to system instability and potential crashes.

6.2.2 Spatial memory error defences

Defending against spatial memory errors has been a central focus of extensive research, resulting in the development of various bounds-checking techniques, categorised into tripwire, pointer-based, and object-based approaches. Tripwire defences, such as AddressSanitizer [218], employ guard blocks (e.g. red zones or invalid memory regions) around memory objects to detect out-of-bounds accesses. These techniques are particularly effective against small-stride contiguous overflow vulnerabilities. However, they are incomplete because they cannot detect cases where pointers go out of bounds but land in another valid allocation. Pointer-based defenses, like SoftBound [193] and CCured [182], use fat pointers — pointers augmented with metadata — to track bounds information for each pointer. This allows for precise checks during dereferencing. While these approaches offer strong security guarantees, their significant performance overheads have limited their use to debugging and testing environments, rather than production systems. Moreover, since fat pointers modify the structure of pointers, changing them from plain values to a combination of a pointer and its associated metadata (e.g. base address and size), they encounter compatibility issues with uninstrumented libraries. Additionally, type casts can corrupt the propagated metadata, further complicating their application. Object-based approaches, such as Baggy Bounds [8], address some of these limitations by storing pointer bounds in a separate metadata space. These methods ensure pointers stay within the bounds of their intended objects by associating metadata with the object itself, rather than individual pointers. While they maintain compatibility with existing C code by preserving the original memory layout, they still face challenges with type casts, sub-object pointers, and incur high overhead.

To modulate performance and further extend memory coverage, OptiSAN [92] intro-

duces a hybrid exploit mitigation approach that dynamically selects between Tripwire defenses, such as AddressSanitizer [218], and object-based bounds checking mechanisms, like Baggy Bounds [8], depending on the specific scenario. This selective approach allows OptiSAN to balance performance and security by choosing the most efficient method for detecting stack memory errors in different contexts. However, it primarily focuses on optimising performance and does not address the other limitations of these approaches, such as compatibility issues with uninstrumented libraries or challenges related to type casting and metadata corruption.

Low-fat pointer approaches have also been proposed to balance performance and compatibility by encoding bounds metadata directly into the pointer representation [80]. By leveraging the unused bits in 64-bit architectures, these approaches embed the base address and size of an object within the pointer itself, eliminating the need for separate metadata structures or shadow spaces. This encoding enables efficient bounds checks during pointer dereferences while preserving binary compatibility and avoiding the performance overheads of fat pointers. Solutions like Delta Pointers [143] have demonstrated the feasibility of this approach to provide spatial memory safety guarantees in the stack. However, low-fat pointers have several inherent limitations, such as memory padding and fragmentation resulting from alignment constraints, challenges with type casting, and an inability to detect intra-object overflows.

More recently, mitigation techniques leveraging the pointer authentication feature introduced in the ARMv8.3 architecture have been proposed to provide spatial and temporal safety guarantees. PACMem [154] and CryptSan [115] use Arm cryptographic Pointer Authentication Codes (PACs) to provide memory safety without requiring fat pointers or intrusive metadata management. However, they still face performance and compatibility challenges, particularly when interacting with legacy code and uninstrumented libraries, which may limit their adoption in real-world systems.

Tagging-based solutions. The advent of memory tagging hardware, especially in ARMv8.5+ architectures, presents a promising hardware-based solution for enhancing spatial memory error defences and reducing performance overhead. Arm Memory Tagging Extension (MTE) operates on a "lock and key" principle, where a 4-bit tag (key) stored in the upper bits of a pointer must match the tag[3] associated with the memory location (lock) to permit access. This mechanism supports both synchronous and asynchronous error detection modes, which determine if an exception should be triggered

[3]In this chapter we will refer to tags and colours interchangeably. Tags can be represented by any finite set of unique items, let them be letters, numbers or colours.

immediately after a tag mismatch or delay it after a context switch. Figure 6.1 and 6.2 show this feature in action. Unlike traditional software-based defences, MTE leverages hardware support to perform these checks, significantly reducing overhead and improving performance. The implementation of MTE typically utilises the Top Byte Ignore (TBI)[4] feature to store tags in the upper unused bits of pointers, with each memory allocation assigned a (unique) tag. However, the native MTE enablement, which relies on assigning random tags via the `IRG` instruction, provides only probabilistic security guarantees [110, 54, 203]. This approach is vulnerable to high collision rates due to the limited entropy of the tag space, potentially allowing attackers to bypass the mechanism. As a result, most MTE-based defences are proposed primarily as sanitisers for testing environments, where the focus is on detecting errors rather than providing robust run-time security guarantees.

Some solutions have attempted to mitigate these issues by proposing exploit mitigation techniques. For instance, ZomeTag [217] introduces a two-layer approach to deterministically assign unique tags to each object by dividing memory into non-overlapping zones using a tripwire mechanism. Each zone can contain up to 16 objects, each with a unique tag. To further reduce performance overhead, StickyTags [101] introduce persistent tagging, where memory re-tagging is not required each time an object is created. Instead, stack and heap memory layouts are organised into per-size-class regions with fixed tags. Despite these advancements, these run-time defences still operate in a sanitisation mode and lack tag integrity, meaning that tags can be manipulated during run-time through arithmetic operations without detection. This vulnerability creates a significant gap that attackers can exploit to bypass such defences, underscoring the need for more robust solutions that ensure full protection against spatial memory errors.

6.3 TAGShield

Stack-based memory vulnerabilities pose a significant security threat, often exploited by attackers to hijack vulnerable applications, alter their behaviour, and extract sensitive data. Although hardware-based techniques have been developed to tackle specific attacks, such as shadow stack [43] for protecting against Return-Oriented Programming (ROP), it is paramount to address the vulnerabilities themselves to prevent more sophisticated attacks (e.g. data-oriented attacks). To this end, TAGShield is arun-time exploit mitigation mechanism designed to address stack-based spatial memory errors

[4]TBI ensures that only the least important 48 bits of a pointers are used to point memory, leaving the rest unused (*ignored*).

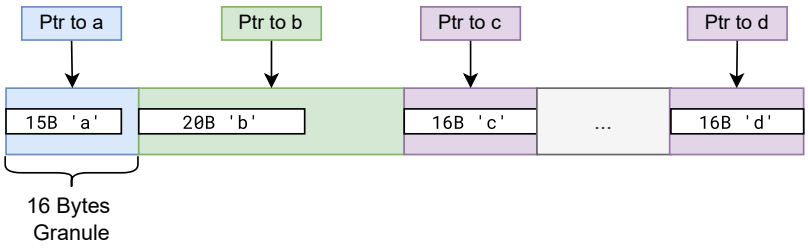


Figure 6.1: Example of how Arm Memory Tagging Extension (MTE) can be used to tag memory. Four memory allocations of different sizes, and their pointers, are tagged with 3 colours (two allocations shares the same tag). Each allocation is padded to respect the 16 Byte MTE granule.

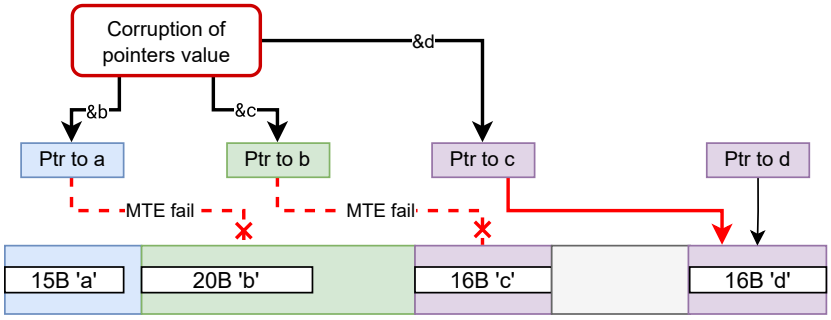


Figure 6.2: Corruption of pointers with the address of other memory areas (`&c` = *address of 'c'*). MTE detects the corruption if and only if there is a colour mismatch.

by combining memory tagging with selective software instrumentation.

It uses an efficient memorycolouring scheme to tag unsafe objects, preventing memory collisions and isolating vulnerable regions. TAGShield also ensures the integrity of pointer tags through lightweight software enforcement, making them resistant to manipulation from pointer arithmetic.

In what follows, we outline our adversary model and then describe the design of TAGShield in detail.

6.3.1 Adversary model

We assume a powerful software-based adversary capable of exploiting any stack-based spatial memory error to mount a wide range ofrun-time attacks, including control-flow hijacking and data-only attacks. The adversary has full control over pointer manipulation and can perform brute-force attacks to modify pointer contents, targeting arbitrary memory locations.

The adversary is assumed to have access to the target application's source code, binary, and compiler information of the target application, enabling detailed analysis of the target's memory layout and behavior. However, the attacker cannot tamper with the source code, binary, or compiler prior to deploying the target binary. Additionally, the adversary may attempt speculative execution attacks to infer memory tags [101], but cannot directly modify MTE tags or alter the MTE configuration, as the adversary operates in user mode and lacks the necessary privileges.

While TAGShield can be extended to address heap-based spatial memory vulnerabilities, such scenarios are considered out of scope for this work and left for future research. Similarly, temporal memory vulnerabilities are excluded from this threat model, as they are addressed by orthogonal defenses specifically designed for such threats [55, 86, 119, 266, 180, 78].

6.3.2 Isolation of safe objects

Memory tagging is a promising security primitive that mitigates memory vulnerabilities by binding pointers to memory using tags. Compared to software-based approaches, Arm MTE transparently checks for tag mismatches at run-time and enriches the Instruction Set Architecture (ISA) with new instructions to explicitly manage the tags (e.g. to assign tags to both memory and pointers). Although MTE's tag checks are relatively low-cost due to their hardware-based implementation, tagging memory can

still introduce non-negligible overhead [219, 101]. To mitigate this, TAGShield selectively instruments only unsafe objects. Specifically, it leverages compile-time static analysis to classify stack allocations into safety categories. Objects and their associated pointers deemed safe are efficiently isolated by retaining the default tag, typically 0x0. In contrast, unsafe objects and their associated pointers are assigned a non-zero tag chosen from the remaining values in the 4-bit tag space. TAGShield not only ensures that unsafe allocations are fully isolated from safe ones by preventing the forgery of the default tag (0x0), but also guarantees that unsafe allocations cannot forge each other's tags.

We define an allocation as memory-safe if all pointer dereferences based on that allocation are either safe or cause the application to terminate. Specifically, we adopt the approach of Kuznetsov et al. [148], which considers an allocation safe if it is accessed exclusively through safe dereferences, i.e. dereferences that only access the memory on which their pointer is based. Conversely, an allocation is unsafe if it is accessed at least once by an unsafe dereference. Intuitively, if all allocations in a program are safe, the program itself is memory-safe.

6.3.3 Deterministic tagging of unsafe objects

The core principle behind ourcolouring scheme is simple: whenever a function is invoked, we explicitly assign a unique tag to each of its stack allocations and propagate this tag to the corresponding pointer. A known limitation of MTE is its low tag entropy (4-bit), which makes it impractical to assign a truly unique tag to every unsafe allocation. Unlike many existing memory tagging implementations [203, 81, 131], we do not rely on random tags. While randomness can deter trivial attacks, it provides no security advantage under a stronger threat model (Section 6.3.1) where an attacker can read memory tags. Instead, we use statically computed tags determined at compile-time and hard-code them within the application. Figure 6.3 illustrates our scheme.

In the function prologue, we assign tags to all unsafe allocations by cycling through the available tag values, ensuring that adjacent allocations receive different tags. The first tag in each function is chosen randomly from the set of non-zero tags, after which subsequent allocations are assigned tags by cycling through the remaining values. These tags are cleared in the function epilogue when the corresponding stack frame is removed. To further reduce the risk of tag collisions across function boundaries, TAGShield's instrumentation pass inserts a 16-byte padding granule (i.e. a red zone) after each stack frame. This prevents the first tag assigned in a new stack frame from inadvertently

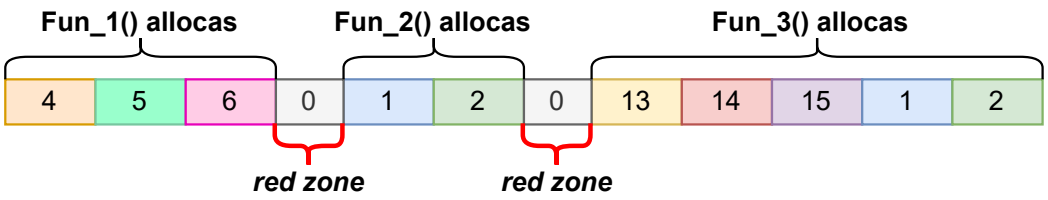


Figure 6.3: Coloring schema. Three nested function cycle the availablecolours starting from a random tag. Between each function, a 0-tagged red zone.

matching the last tag of the previous frame. This red zone is assigned the default tag 0, ensuring that no unsafe pointer can access it.

While alternating tag values mitigate contiguous (i.e. linear) memory overflows, they do not protect against non-contiguous (i.e. non-linear) overflows, which often arise from pointer arithmetic - a common pattern in real-world applications. An adversary can exploit this limitation to manipulate tags to access unintended memory objects without detection. Addressing this issue is a key objective of TAGShield, as detailed in Section 6.3.4.

Furthermore, due to the low entropy of tag values (only 4 bits), multiple live allocations may share the same tag (orcolour). This could allow a vulnerable pointer to access other allocations (memory objects) with matching tags if pointer arithmetic modifies only the pointer's address while leaving the tag intact. Tag collisions are an inherent limitation of memory tagging approaches, and since MTE does not inherently bind tag values to specific addresses, this issue remains unresolved in all pure MTE-based approaches including TAGShield. One potential solution is to integrate ARM Pointer Authentication Code, though this is beyond the scope of this paper. However, since TAGShield mainly targets unsafe allocations, which represent a small subset of all allocations, the risk of such exploitation is significantly reduced.

Selective tag checks

To further improve the performance of TAGShield, we optimize memory tagging by differentiating between types of memory accesses rather than applying tagging uniformly to all unsafe allocations. Stack memory can be accessed in two ways: via stack pointers (SP) or other pointers. SP-based accesses use the stack pointer with a static offset to reference specific memory locations. Given the adversary model described in Section 6.3.1, instructions using SP-based accesses are considered *write-safe* because they directly target fixed memory locations, leaving no opportunity for an adversary to redi-

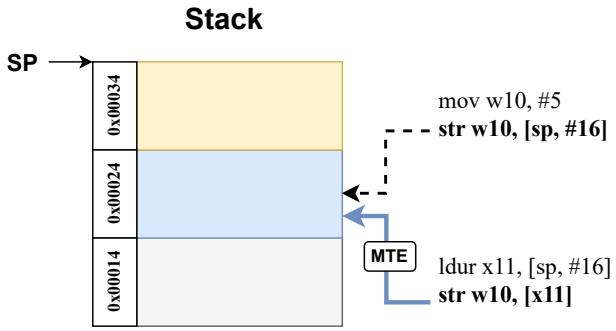


Figure 6.4: Two types of memory access in Arm assembly. The first one uses an SP-based address, the second one uses a pointer-based address. Only the latter is protected with TAGShield.

rect them to unintended addresses. In contrast, pointer-based accesses rely on pointers stored in memory, making them inherently more susceptible to exploitation. While pointers enhance program flexibility by enabling operations such as pointer arithmetic and dynamic array accesses, they also introduce security risks. An adversary could exploit a memory vulnerability to manipulate a pointer stored on the stack, causing subsequent pointer-based accesses to reference unintended memory locations. Therefore, pointer-based accesses are classified as *write-unsafe*: an adversary who gains control over a pointer could leverage it to corrupt arbitrary memory locations. Since SP-based accesses cannot be hijacked to target unintended locations, TAGShield focuses exclusively on protecting pointer-based accesses to unsafe allocations. Figure 6.4 illustrates these two types of accesses.

6.3.4 Enforcing integrity of tags

In the case of MTE, a corrupted pointer has a theoretical probability of $^{14}/_{15}$ of triggering a tag mismatch, with a $^{1}/_{15}$ chance of going undetected. To mitigate tag collisions, several works [203, 217, 101] alternate tag values in memory, ensuring that contiguous overflows trigger a tag mismatch. However, this approach does not fully eliminate tag collisions. Non-contiguous buffer overflows can still target distant memory locations that share the same tag, without detection.

In practice, pointer arithmetic, which is common in programming constructs, not only allows adversaries to manipulate addresses to target allocations with matching tags (as shown in Figure 6.2) but also enables them to forge tags themselves, as MTE does not protect tags. Recall that tags are simply stored alongside the pointer's content.

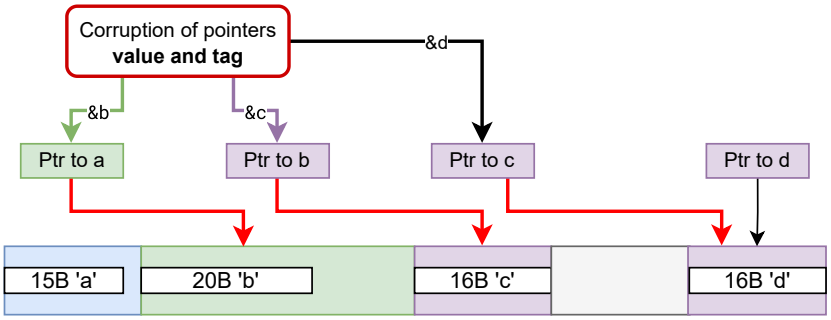


Figure 6.5: Corruption of both pointers and tags to circumnvent MTE.

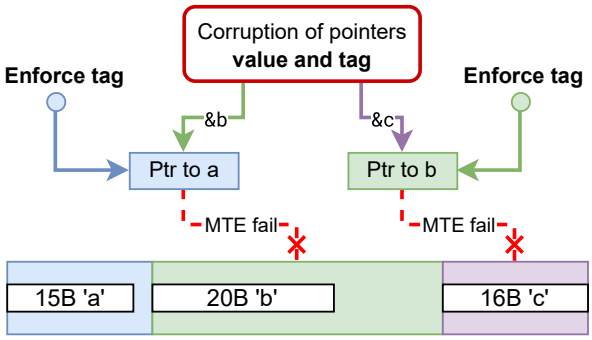


Figure 6.6: Example of attacker corrupting both tag and value of the pointers with TAGShield enabled. This restores the tag before the pointers are used.

For instance, consider a scenario where an adversary gains full control over a pointer, `x`, which points to `X[1]`[5]. If the adversary's goal is to overwrite the memory locations `Y[2]` and `W[1]`, they can do so as follows. To access `W[1]`, it suffices to inject `W`'s address into `x`, as illustrated in Figure 6.2. However, to access `Y[2]`, the adversary must also modify the tag stored in `x`, as shown in Figure 6.5. By doing so, the adversary effectively gains an arbitrary memory read/write primitive, completely bypassing any MTE check.

As mentioned earlier, defending against the first case, where only pointer addresses are manipulated to target allocations with matching tags, is beyond the capability of any MTE-based approach. Therefore, TAGShield focuses on addressing the latter case, which is largely overlooked by existing approaches. Mitigating this issue eliminates a significant attack surface. To achieve this, TAGShield enforces that a pointer can only be dereferenced if its tag matches the tag assigned by the employedcolouring scheme at compile-time, regardless of whether the embedded tag has been corrupted. Figure 6.6 illustrates TAGShield's tag integrity mechanism in action.

[5]We use capitalised letters to indicate memory allocations and square brackets to indicate the tag of the allocation.

Mechanism

Upon accessing a tagged pointer, TAGShield ensures that its tag is authentic. Enforcing tag integrity, however, is not always straightforward and depends on the type of pointer. In particular, we distinguish between two types of pointers: *root level* and *non-root level*. A pointer is considered root level when it directly references a stack allocation without any prior dereference. In LLVM IR (described in Appendix B.1), where TAGShield's passes are implemented, every alloca instruction generates a root level pointer (e.g. '`%a`' and '`%ptr`' in Listing 6.1). Conversely, non-root level pointers are derived by pivoting from another root or non-root level pointer. In C/C++, this is typically expressed through pointer dereferencing, forming a chain of memory accesses. For instance, the operation `*(ptr)` consists of two memory accesses:

- The first access involves the root level pointer, `*(ptr)`, namely `%ptr` in Listing 6.1.
- The second access pivots from the root pointer `ptr` by dereferencing a non-root level pointer `*(ptr)`, which corresponds to '`%0`' in Listing 6.1.

Notably, all non-root level accesses must originate from root level accesses: first, the pointer value (root level) is loaded, and only then it is dereferenced (non-root level).

```
//Root level pointers
%a = alloca i32, align 4
%ptr = alloca ptr, align 4

//Non-root level pointers
%0 = load ptr, ptr %ptr, align 4
```

Listing 6.1: Examples of root level and non-root level pointers.

For every unsafe alloca instruction that generates a root level pointer, TAGShield's plugin in the compiler toolchain generates and embeds a distinct tag. The integrity of this tag is safeguarded by hard-coding it in the preamble of the corresponding stack frame and enforcing it with every access, as shown in Listing 6.2. In other words, whenever the pointer is accessed, we reset its tag to the authentic one computed at compile-time, ensuring that it remains persistent throughout the pointer's lifecycle.

Non-root pointers, however, present additional challenges since their authentic tags may not be known at compile-time. Consider the code listing *App Instructions* in Figure 6.7, which includes three variables (`int x`, `int y`, `int * ptr`) tagged statically with red, blue and green, respectively. Accesses to the root level pointer `ptr`, such as ①

```
call @enforceStaticTag(%a, #tag)
store #40, ptr %a, align 4
```

Listing 6.2: Enforcement of a root level tag for a variable, e.g. `%a = alloca i32`.

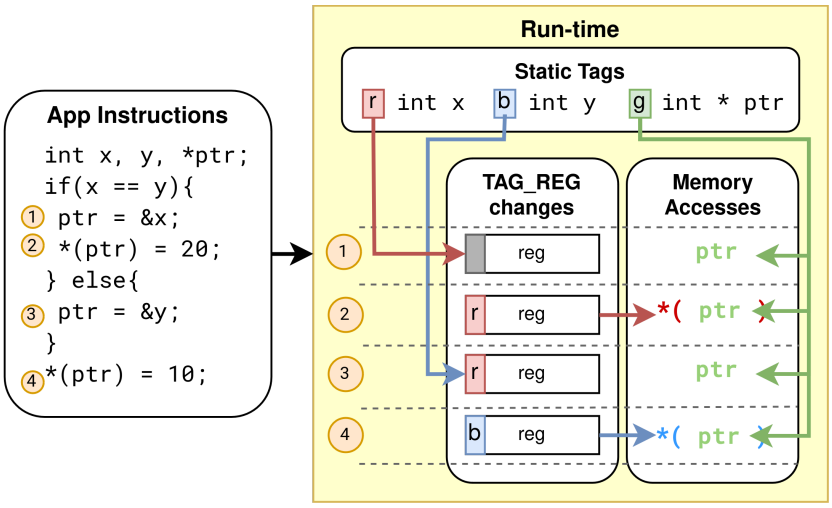


Figure 6.7: Representation of how pointer accesses are protected by TAGShield. When a pointer content is changed, the source tag is updated in `TAG_REG`. Each root level access is then protected with the static tag, while each non-root access is protected with the tag saved on `TAG_REG`.

and ③, are handled statically by enforcing its green tag. Conversely, determining the tag for the non-root pointer `*(ptr)` at ④ is non-trivial. `*(ptr)` could be pointing to either `int x` or `int y`, each having a different tag. In general, when a pointer's content changes to reference a new variable, the tag must match that of the referenced variable upon dereference.

Interestingly, since MTE tags are embedded in pointers without integrity guarantees, altering a pointer's content automatically updates its tag to match the source pointer. Nevertheless, TAGShield must track these tag changes to enforce tag integrity at runtime. Specifically, at any point during execution, TAGShield must determine the correct tag for non-root level pointers. We call the set of all non-root level pointers' tags *TAGShield state*.

TAGShield tracks its state using dedicated registers (`TAG_REGs`) and software instrumentation. For each non-root pointer in a function, TAGShield reserves 4 bits in these dedicated registers. For instance, given the C variables `int a, *pa, **ppa;`, TAGShield reserves 4 bits each for `*(pa)`, `*(ppa)` and `*(*(ppa))`. Whenever any of these non-root pointers is assigned an address, the corresponding register entry is up-

dated with the tag of the source pointer. For the code listing in Figure 6.7, TAGShield reserves 4 bits to track the tag of `*(ptr)`, updating it at lines ① and ③.

By tracking the tags of non-root pointers, TAGShield instruments each non-root dereference to enforce the authentic tag, fetching it from the TAGShield registers as shown in Listing 6.3. During compilation, TAGShield determines the location of each non-root pointer's tag in the registers, allowing efficient run-time enforcement.

```
call @enforceStaticTag(%ptr, #tag)
%0 = load ptr, ptr %ptr
call @enforceDynamicTag(%0, #tag_index)
store #40, ptr %0, align 4
```

Listing 6.3: Enforcement of tags for a pointer dereference. The first root level access to the variable is protected with a static tag. The second non-root access is protected by enforcing the tag saved on `TAG_REG`.

Inter functions operations

Tags are typically local to a function and remain active for its duration. Consequently, TAGShield assigns static tags to both pointers and memory during the function prologue and removes these tags in the epilogue, ensuring memory protection throughout the function's lifetime. Similarly, the TAGShield state must be preserved and remain valid for the entire function execution, allowing subroutines to manage their own TAGShield state.

To achieve this, TAGShield spills the content of the dedicated registers to the stack in the function prologue, treating it as a safe allocation. This ensures the caller function's TAGShield state is saved before being overwritten. The state is then restored prior to returning. Importantly, this backup and restore operation is only performed if the invoked function requires a TAGShield state, meaning it involves unsafe dereferences that need protection.

6.3.5 Limitations

Similarly to other works in the literature [54, 156, 101, 217, 251, 197, 168], TAGShield does not protect sub-objects in structs, treating them as part of the same allocation and tagging them accordingly.

Furthermore, we currently limit our tag integrity schema to coope with *pointer*

aliases[6] in complex programs. Consider the code listing in Figure 6.8 where on line ② an alias of '`ptr`' is created and stored it in '`pptr`', making '`pptr`' the alias of '`ptr`'. Once the alias is created, it can reference the original pointer, as seen on line ③ where it modifies the content of `ptr`'. TAGShield tracks the tags for non-root pointers. Thus, on ③ TAGShield state should be updated to ensure that dereferences of both '`*(ptr)`' and '`*(*(pptr))`' (e.g. on ⑤ and ④, respectively) enforce the correct tag, as both points to the same memory area. In other words, after line ③, the tag states for both non-root pointers should remain synchronised.

Failure to synchronise pointer aliases when making tags persistent can result in false negatives during MTE checks. For instance, on line ⑤, enforcing the incorrect red tag on `*(ptr)` (as per the state saved on ①) occurs becuase TAGShield state was not updated on line ③.

Detecting pointer aliases effectively remains an open research challenge. Moreover, maintaining synchronised states requires significant software instrumentation overhead. For these reasons, after a pointer is aliased, TAGShield defaults to its bare colouring schema for protection and ceases enforcing tag integrity for the aliased pointer and its aliases.

6.4 Implementation

TAGShield is an advanced memory tagging schema that requires hardware support at run-time for storing memory tags and performing tag checks. Although other hardware memory tagging architectures exist — such as SPARC ADI [158], lowRISC [82], and the work-in-progress RISC-V memory tagging [108] — we chose Arm MTE due to its growing popularity and increasing adoption in commercial products.

Arm MTE automatically performs tag checks on memory accesses when either the memory or the pointer is tagged. However, tagging memory and pointers must be implemented in software using the new MTE instructions. Additionally, as described in Section 6.3.4, the tag integrity schema in TAGShield requires explicit software instrumentation to ensure tag persistence and to maintain an up-to-date TAGShield state. To achieve this, we leverage LLVM compiler toolchain version 18 [201], widely recognised in industry and academia for its modularity and extensibility (detailed in Appendix B.1).

TAGShield's implementation includes two LLVM components: a front-end pass and

[6]Pointers whose address is stored into another pointer.

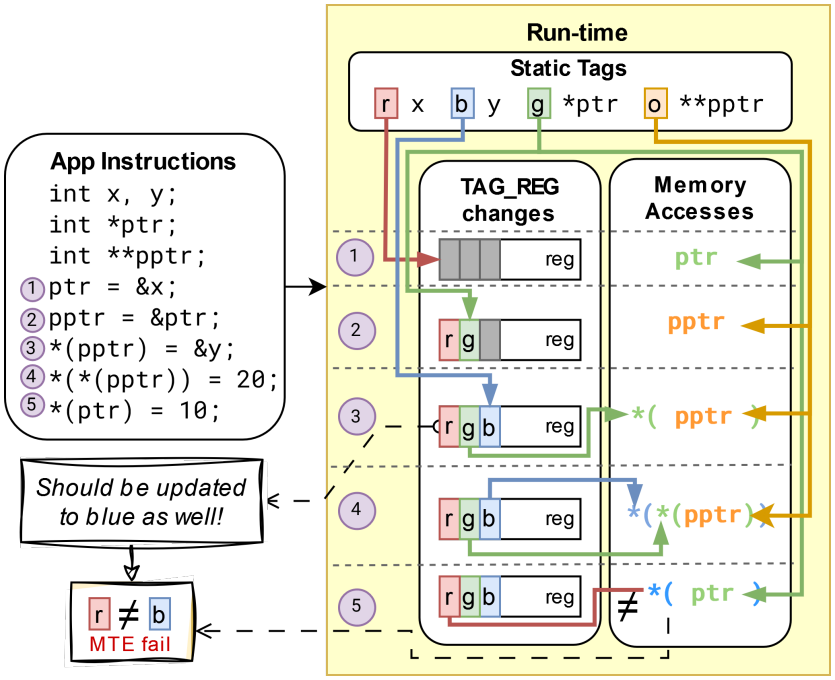


Figure 6.8: Tag integrity issue with pointer aliasing. After the `ptr` alias created at ②, instruction ③ should change the TAGShield state for both `*(ptr)` and `*(*(pptr)`. Failing to do so will cause an MTE failure on ⑤.

a AArch64 back-end extension. The front-end pass operates at the LLVM Intermediate Representation (IR) level to analyse code (e.g. for identifying pointer aliases) and to instrument IR with custom instructions, referred to as *intrinsics*. By operating at the IR level, TAGShield gains portability, enabling potential extension to other tagging architectures. Examples of TAGShield IR instrumentation are shown in Listings 6.2 and 6.3.

Our front-end pass was implemented within the new LLVM pass manager and consists in ∼1500 Lines of Code (LoC). To perform the various analysis, we leverage the *AAResultsWrapperPass*, *StackSafetyGlobalInfoWrapperPass*, *TargetLibraryInfoWrapperPass* and *MemoryDependenceWrapperPass* passes. To identify safe memory allocations, TAGShield utilises LLVM's *StackSafetyAnalysis* pass [205], a dedicated component already integrated into LLVM.

Although IR is an architecture-independent representation, the final TAGShield instrumentation is specific to the AArch64 architecture, for which MTE is implemented. Consequently, TAGShield includes custom back-end functions to convert instrumented IR into AArch64 assembly. These functions handle the *lowering* of intrinsics, i.e. transforming custom instructions into architecture-specific assembly code. While our implementation incorporates various optimisations to use the most efficient Arm instructions,

most intrinsics can be generalised to two primary instructions: `movk` and `bfm`. The `movk` instruction sets 16 bits of a register, while `bfm` moves a specified bit sequence from one register to another. Listing 6.4 demonstrates how both static and dynamic tag enforcement can be lowered into assembly. In total, our back-end extension consists in ∼700 LoC.

TAGShield is an open-source project, with its repository available[7] at [30].

```
ldr x8, [sp, #8] ; root-level address
movk x8, #tag, 56 ; set tag
ldr x9, [x8]  ; first-level address
bfm TAG_REG, x10, #immr, #imms
bfm x10, x9, #immr, #imms
str x11, [x9] ; use the pointer
```

Listing 6.4: Enforcement of tags for non-root level dereferences. The number of `bfm` operations and their parameters vary depending on where the tag is saved on `TAG_REG`.

6.5 Evaluation

Although Arm released MTE in 2019, its adoption remains limited, with few CPUs and devices supporting it at the time of writing. While emulating the MTE extension is a common practice in research [217, 168, 251, 197, 54, 156], we opted to test TAGShield on real MTE-capable hardware to provide a more accurate assessment of its performance. Specifically, we used a Google Pixel 8 equipped with the Google Tensor G3, which comprises 4 Arm Cortex-A715 and 4 Arm Cortex-A510 cores, all of which support MTE. The tests were conducted on a rooted[8] device within a Docker container running Gentoo Linux [91]. We chose Gentoo over the native Android environment for its compatibility with the full standard C library (stdlib).[9]

To comprehensively evaluate TAGShield's performance across various scenarios, we performed an extensive evaluation using two popular benchmarks: the NBench-byte [167] micro benchmark suite and SPEC CPU 2006 [227]. Additionally, we evaluated three real-world application test cases to understand TAGShield's behavior in production environments: a stand-alone SHA crypto engine [252], the Nginx [232] web server and an OP-TEE [249] AES Trusted Application (TA). For each case, we compare the

[7] The code will be uploaded upon acceptance of our submitted paper.
[8] A device running a modified version of the kernel with unrestricted root access.
[9] Our tests use the original stdlib, without any TAGShield instrumentation.

Table 6.1: Statistics about the various benchmarks and real-world applications that are tested with TAGShield.

	Benchmarks	Code memory size (original)	Code memory size (TAGShield)	Overhead %	Data memory size (original)	Data memory size (TAGShield)	Overhead %	Total objects	Unsafe objects	Ratio %
SPEC CPU 2006	401.bzip2	106.2 KB	105.2 KB	-3.40%	4000 B	4088 B	2.20%	395	55	14%
	403.gcc	5455 KB	5407 KB	-0.88%	160.5 KB	160.6 KB	0.04%	5978	859	14%
	429.mcf	19.5 KB	21.4 KB	9.82%	696 B	768 B	10.34%	44	11	25%
	433.milc	163 KB	169 KB	3.72%	1993 B	2057 B	3.21%	884	202	23%
	444.namd	472 KB	457 KB	-3.18%	1144 B	1208 B	5.59%	9578	573	6%
	445.gobmk	2406 KB	2422 KB	0.68%	1942 KB	1942 KB	0.00%	3875	1019	26%
	447.dealII	4676 KB	5126 KB	9.63%	66.8 KB	66.9 KB	0.14%	57632	22733	39%
	450.soplex	506 KB	514 KB	1.62%	8984 B	9080 B	1.07%	2039	535	26%
	456.hmmer	409 KB	416 KB	1.68%	8768 B	8808 B	0.461%	1374	279	20%
	458.sjeng	205.8 KB	207.8 KB	0.94%	8344 B	8408 B	0.77%	489	133	27%
	462.libquantum	51.6 KB	55.5 KB	7.59%	748 B	820 B	9.63%	286	99	35%
	470.lbm	876 KB	867 KB	-0.99%	680 B	752 B	10.59%	66	17	26%
	473.astar	62.7 KB	66.2 KB	5.68%	860 B	948 B	10.23%	354	82	23%
	482.sphinx3	238 KB	243 KB	2.18%	3696 B	3752 B	1.52%	561	131	23%
	NBench-byte	61 KB	64.5 KB	5.64%	2144 B	2200 B	2.61%	59	59	100%
	Nginx	761 KB	760 KB	-0.18%	35.4 KB	35.5 KB	0.24%	2054	372	18%
	SHA2-512	10.7 KB	10.8 KB	1.10%	644 B	644 B	0%	48	10	21%
	AES-TA	4685 B	4589 B	-2.05%	0 B	0 B	-	19	2	11%

performance of the original binaries against those protected by TAGShield,[10] measured over 10+ different test runs. Figure 6.12 provides an overview of all the performance results, while Table 6.1 summarizes the memory overhead and the number of objects protected by TAGShield for each benchmark and real-world application. Interestingly, it can be seen that memory overhead is negligible in most cases with some applications yielding a smaller code size after TAGShield instrumentation. This is due to some change in the stack access patterns when introducing memory tagging, which in some cases reduces the code size, albeit increasing the access latency.

6.5.1 Benchmarks

Nbench-byte. The NBench-byte benchmark is a collection of algorithm-level tests that measure the performance of a system's Central Processing Unit (CPU), Floating Point Unit (FPU) and memory system. We chose this suite because it is open-source and it has already been used to evaluate MTE in previous work [251]. Our results show that TAGShield introduces minimal overhead in these benchmarks, with a geometric mean of just 2.96%. Notably, nearly all tests exhibited negligible performance degradation (less than 6%), with one exception: the Huffman coding test, which incurred a moderate overhead of 48.18%. An overview of the performance overhead across these benchmarks is presented in Figure 6.9.

[10]We configured MTE in SYNC mode, thus triggering an exception on each tag mismatch.

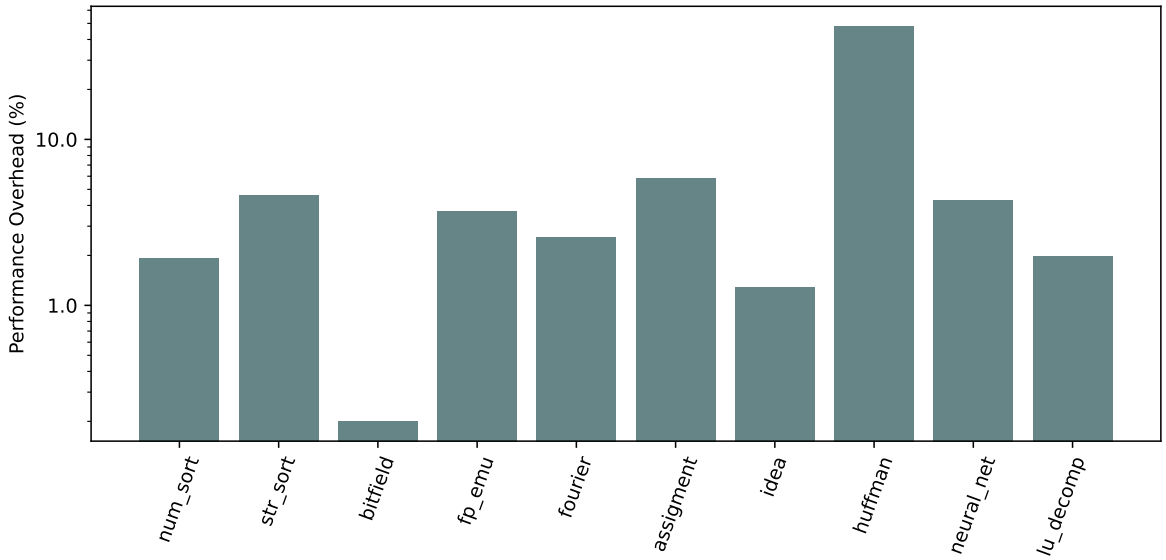


Figure 6.9: Performance overhead for the NBench-byte benchmark suite.

SPEC CPU 2006. The SPEC CPU test suite is a widely recognized benchmark comprising compute-intensive applications designed to evaluate CPU performance. Various versions, from v89 to the latest v2017, have been extensively used in the literature as a standard performance metric, with several related works [54, 217, 101] adopting it as well. In our evaluation, we use the older yet still relevant v2006 instead of v2017, as the latter is designed for high-end devices with more powerful resources. Notably, most of its tests require 16 GB of RAM, exceeding the 8 GB available on the Pixel 8. SPEC CPU 2006 includes over 30 test cases written in various programming languages. Given TAGShield's focus on memory safety, we restricted our evaluation to C and C++ test cases. Our results reveal a geometric mean overhead of 23.97%, with a minimum of 0.69% (*429.mcf*) and a maximum of 93.28% (*447.dealII*). Figure 6.10 provides a detailed summary of the evaluation.

6.5.2 Real-world applications

While benchmarks like SPEC CPU 2006 and NBench-byte highlight TAGShield's impact on compute-intensive workloads, they often polarise results and may not accurately reflect real-world scenarios. To assess TAGShield's suitability in production environments, we tested its performance on three representative applications.

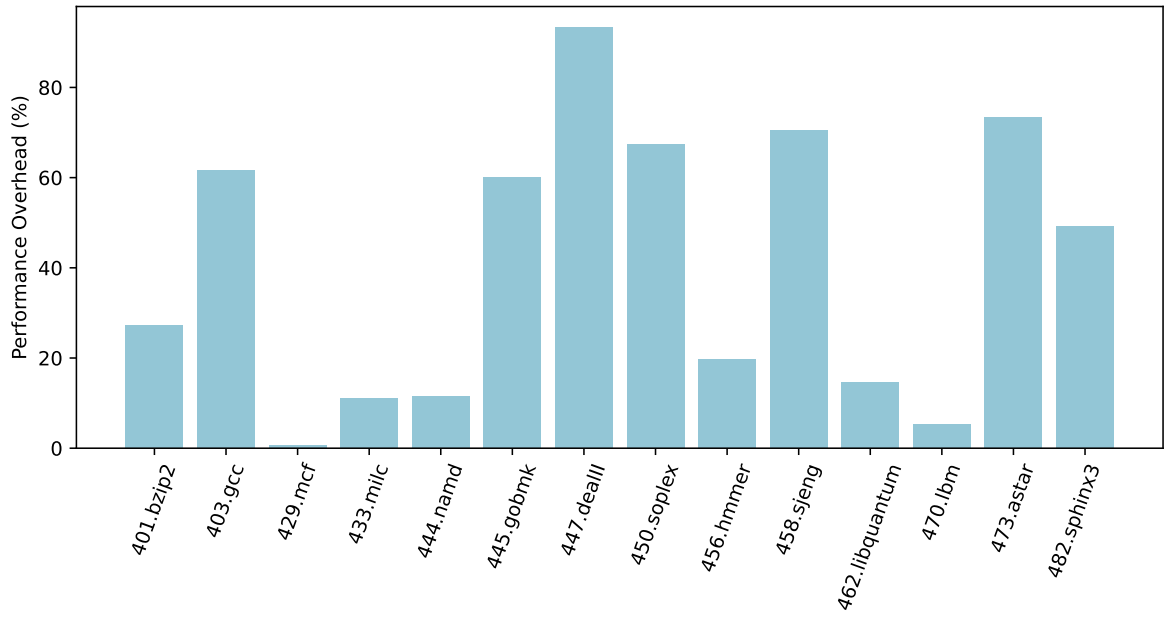


Figure 6.10: Performance overhead for the CPU SPEC 2006 benchmark suite.

SHA crypto engine. Cryptographic engines are critical components of Trusted Computing and provide a compelling real-world use case. Protecting these systems against attackers is of utmost importance. We evaluated TAGShield on an open-source SHA2-512 cryptographic engine [252] by computing digests for datasets of varying sizes (1 MB, 10 MB, 100 MB, and 1 GB). The observed performance overheads were 4.33%, 4.46%, 3.94%, and 4.52%, respectively, indicating consistent and low overhead across different input sizes.

Nginx. Nginx, one of the most widely used web servers, powers 33.8% of known websites [258], making it an ideal candidate to evaluate TAGShield's real-world performance. We measured the average delay of an Nginx instance serving a static page across 12 threads, handling 400 open connections over a 30-second duration. The results showed an average overhead of just 0.06%, demonstrating that TAGShield imposes negligible performance degradation in this scenario.

A Trusted Application

Trusted Execution Environments (TEEs) play a critical role in modern computer systems, handling security-sensitive operations that the entire system relies on. While TEEs offer strong protection for the software running within them, they are not immune to bugs and vulnerabilities that can be exploited from the Rich Execution Environment

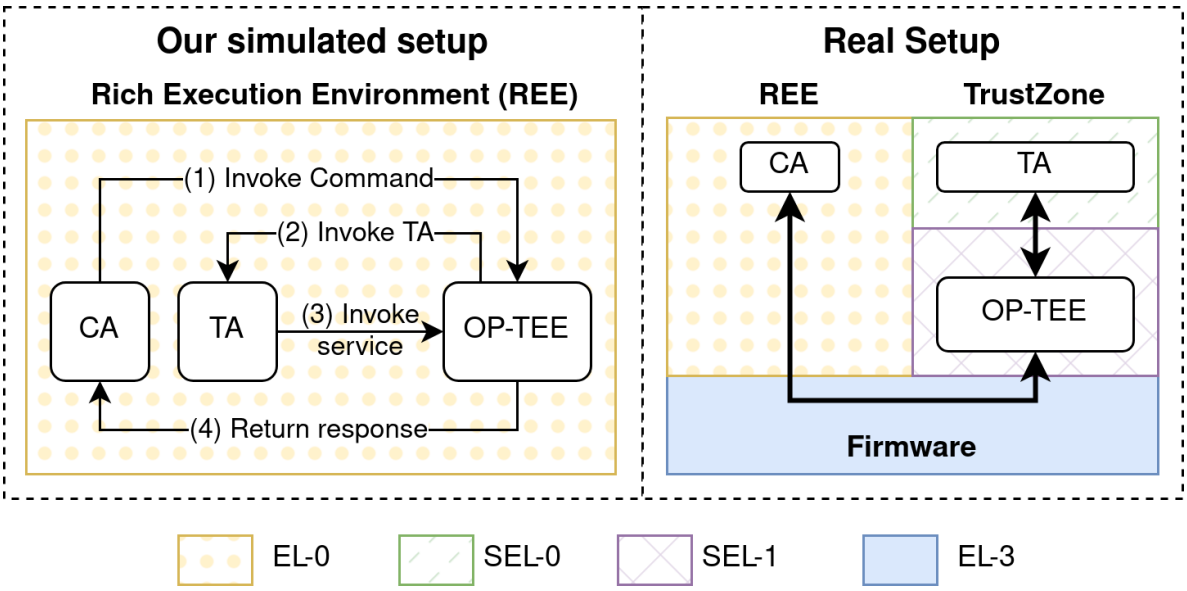


Figure 6.11: TAGShield setup for the evaluation of an OP-TEE Trusted Application (TA). On the left our setup, with Client Application (CA), TA and OP-TEE running inside the Rich Execution Environment (REE). On the right, a real setup where the CA runs in REE and the TA with OP-TEE in Trustzone.

(REE). Over the years, multiple CVEs have been reported affecting TEEs, including their operating systems (TEE OSes) and Trusted Applications (TAs), demonstrating that simply having a TEE does not guarantee security [48]. Given the potentially severe consequences of TEE vulnerabilities, incorporating additional security layers — such as TAGShield — can significantly enhance the overall security of the system.

To evaluate the feasibility of our approach, we tested the impact of TAGShield on a TrustZone-based system running the open-source OP-TEE OS [249]. Instead of enabling TAGShield across the entire TEE OS, we focused on a specific Trusted Application (TA) responsible for AES encryption.

Like many TrustZone-based systems, OP-TEE functions as an operating system that manages multiple TAs, all of which run in the Secure World and are accessible from the REE via GlobalPlatform APIs[11] [94]. Figure 6.11 illustrates the interaction between system components. Precisely, when the REE requires a specific service, e.g. the encryption of a buffer, the following steps occur:

1. The Client Application (CA), i.e. an application appositely crafted to interact with a specific TA, sends a request to the OP-TEE core component, a.k.a. OP-TEE OS. To do so, it invokes one or more of the GlobalPlatform Client APIs,

[11]This is a standard that regulates the interactions between TEE and REE, and between the various components of the TEE.

for instance `TEEC_InitializeContext()` and `TEEC_InvokeCommand()`. Invoking such command triggers a Secure Monitor Call (SMC) to switch to the Secure World, transition handled by the Secure Monitor running in EL3.[12]

2. The OP-TEE core, running in S_EL1, parses the request from the CA, it converts the input parameters into a format that the TA can understand, and forwards the request to the relevant TA using the GlobalPlatform Core APIs, e.g. `TA_InvokeCommandEntryPoint()`. By invoking the TA, the core switches to S_EL0.

3. The TA processes the request and, possibly, it invokes some internal OP-TEE core services to complete the operations. This is once again done through the GlobalPlatform Core APIs, e.g. `TEE_CipherUpdate()`, and brings the execution back to SEL-1. Interestingly, most TAs in OP-TEE and GlobalPlatform-compliant systems are wrappers around the services provided by the OP-TEE core.

4. The OP-TEE core performs the requested services and then returns the execution back to the TA.

5. After the TA has completed its operations, it sends the response back to the OP-TEE core, which forwards it to the CA, passing once again through the Secure Monitor. The response is then processed by the CA, which can finally return the result to the user.

Since deploying OP-TEE on the TrustZone of our Pixel 8 is virtually impossible, we created a mock-up OP-TEE setup running entirely in the REE. Our setup consists of three components: the TAGShield-instrumented TA, a minimal OP-TEE core, and a CA to invoke the TA's AES encryption service. Unlike a real deployment where OP-TEE and the TA run in TrustZone, respectively at SEL-1 and SEL-0, all components in our setup execute at the user level (EL0) in the REE, as shown in Figure 6.11.

To measure the performance impact of TAGShield, we configured the CA to request 300 AES encryptions of a 4 KB buffer, and measured the time it took for the TA to respond to such requests. Our results show that TAGShield introduces an average performance overhead of 6.00%. Notably, this setup simplifies system-level interactions by avoiding privilege level transitions and Secure/Non-Secure world switches. While this means our measurements do not perfectly reflect real-world performance, the reported

[12]Exception Level (EL) is used to refer to the exception levels in the Normal World and Secure Exception Level (S_EL) for those in the Secure World.

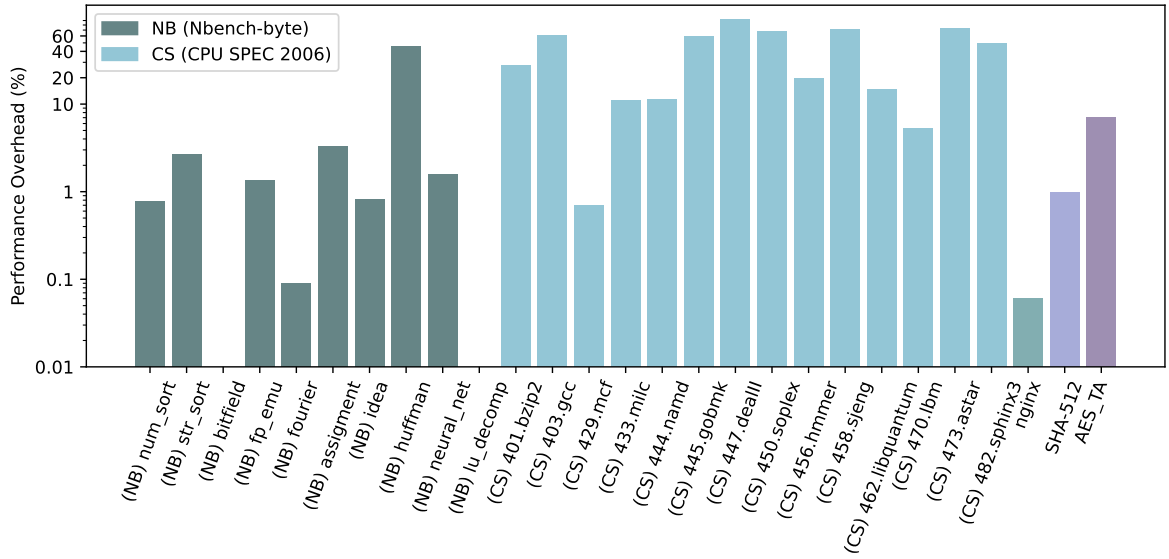


Figure 6.12: Overview of the results for the performance evaluation of all TAGShield-instrumented binaries. We grouped NBench-byte (NB), CPU SPEC 2006 (CS) and our 3 real-world applications.

overhead serves as a lower bound, as these system interactions — being outside the scope of TAGShield instrumentation — would remain unaffected by it.

6.6 Discussion

6.6.1 Security analysis

Since its introduction by Arm, MTE has been integrated into several production-ready environments, including Scudo [204], glibc [81], KASAN [131], Chrome [245], and LLVM [203]. It has also been employed in numerous research projects [251, 217, 101, 156]. However, despite its promise, current solutions are not impervious to attacks and have been demonstrated to exhibit vulnerabilities [197, 136].

A significant attack vector involves tag corruption, where an attacker manipulates the tag to transform a pointer into an arbitrary memory read/write primitive. TAGShield addresses this critical security gap by providing lightweight tag integrity protection, as described in Section 6.3.4.

To evaluate the effectiveness of our technique, we tested it on a sample application [30] and show that TAGShield can thwart powerful attackers that have full control over a pointer. The application implements a basic substring extraction function, which requests a string and an index from the user before printing the requested substring.

As shown in Listing 6.5, the application contains a non-linear buffer overflow vulnerability. An attacker can exploit this by providing an excessively large index, corrupting the `name` pointer. This corruption allows access to illicit memory locations, potentially enabling arbitrary memory read or write primitives. Such primitives can serve as stepping stones for traditional control-flow attacks, such as Return-Oriented Programming (ROP), or advanced data-oriented attacks such as Direct Data Manipulation (DDM), Data-Oriented Programming (DOP) or Block-Oriented Programming (BOP).

To evaluate TAGShield, we modeled two types of attackers:

1. Simple Attacker: corrupts the pointer to redirect it to a different memory location.
2. Sophisticated Attacker: corrupts both the pointer and its associated tag, forging a valid tag for the target memory address.

We performed these attacks on three versions of the application binary, each employing a different level of protection.

Unprotected binary. In the first scenario, the application was tested without any memory tagging. As expected, both the simple and sophisticated attackers successfully pivot the corrupted pointer, accessing arbitrary memory locations undetected. This highlights the severity of stack-based memory vulnerabilities in binaries lacking the proper defences.

MemTagSanitizer-hardened binary. Next, we tested a binary protected by MemTagSanitizer [203], a state-of-the-art memory tagging solution integrated into LLVM. This defence manages to thwart the simple attacker, whose pointer corruption triggers an MTE tag mismatch. However, the sophisticated attacker manages to forge the tag of the target memory address, successfully bypassing the MTE protection and writing/reading an arbitrary memory location. This empirical result demonstrate how tag forgery poses a serious threat to MTE-based approaches.

TAGShield-hardened binary. Finally, we evaluated a binary instrumented with TAGShield. As with MemTagSanitizer, the simple attacker is thwarted by the memory tagging schema. However, unlike the previous case, TAGShield effectively neutralises the sophisticated attacker. Even though the attacker corrupts both the pointer and its tag, TAGShield's tag integrity schema mechanism restors the authentic tag before granting access to the corrupted pointer.

Through this evaluation, we demonstrated that TAGShield successfully prevents tag forgery, thereby mitigating sophisticated attacks that exploit MTE's inherent weaknesses. By restoring the correct tag and validating tag integrity, TAGShield provides robust protection against critical vulnerabilities, effectively thwarting attacks and enhancing the overall security of memory tagging systems.

```
fgets(input, sizeof(input), stdin); //Get an input from standard
    input
offset = strtoul(input, NULL, 10); //Convert the given input into
    a number
printf("Substring: '%s'\n", &name[offset]); //Print the substring
    of 'name' starting from the computed offset.
```

Listing 6.5: Example of non-linear buffer overflow vulnerability that can be defended with TAGShield.

Juliet evaluation

To assess the correctness of our implementation, we evaluated it using a subset of the Juliet C/C++ 1.3 Test Suite from NIST [40]. Juliet is a collection of common vulnerabilities, extensively used in the literature to evaluate the effectiveness of memory defence techniques. The test suite comprises 64,099 test cases, organized hierarchically by Common Weakness Enumeration (CWE) categories, specific vulnerabilities, and unique variants. For our evaluation, we focused on CWE-121 (Stack-Based Buffer Overflows) and excluded irrelevant tests, such as 32-bit-only bugs, Windows-specific bugs, and non-deterministic cases, resulting in 2,362 variants.

As shown in Table 6.2, our implementation successfully triggered a tag mismatch in 47 cases (1,925 variants), with no missed detections. Additionally, 9 cases (359 flow variants) were mitigated by the TAGShield allocator, which rounds up allocation sizes to the MTE tag granularity, preventing off-by-one bugs in non-multiple-sized allocations.

Notably, this security evaluation does not measure the full impact of TAGShield, as the Juliet test cases do not include tag corruption attacks.

Table 6.2: Juliet test cases: $N(M)$ represents N unique cases with M flow variants.

Allocator Fix	TAGShield Detect	TAGShield Miss
9 (359)	47 (1925)	0 (0)

6.6.2 An alternative design: detecting tag forgery

Section 6.3.4 discussed how TAGShield tracks various tags at run-time to ensure their persistence, thereby providing lightweight tag integrity. This design guarantees that any misuse of a pointer or attempts to corrupt its tag are detected by hardware MTE checks. Arm MTE supports two primary modes: synchronous and asynchronous. The synchronous mode triggers an exception immediately upon a tag mismatch, while the asynchronous mode sets a mismatch bit in a control register, triggering the exception only during a context switch. These modes enable memory tagging to function as either a detection mechanism or a preventive measure. In our evaluation, we utilised the synchronous mode to prevent any unauthorised memory access.

However, the current design cannot detect tag forgery when MTE operates in either mode. TAGShield ensures tag persistence at run-time, so any corruption is overwritten by the authentic tag without leaving evidence. To enable tag forgery detection rather than prevention, we could modify the design to compare pointers' tags against those stored in TAGShield's state. Any mismatch could then trigger an exception, signalling tag forgery rather than generic pointer corruption. This alternative design, however, would likely impose a greater performance overhead due to the additional instrumentation required.

6.6.3 Future work

Our TAGShield design and implementation currently serve as a functional proof of concept. The work presented in this thesis can be further refined and extended to address its limitations and enhance performance. Below, we outline potential directions for future work:

Expanded memory coverage. TAGShield is a memory exploitation mitigation technique designed to address stack-based vulnerabilities, given their prominence compared to heap-based ones. However, the design can be extended to cover heap allocations. Specifically, the memory allocator could be modified to reserve a tag upon request, passing the static tag to the malloc function to ensure root-level tag persistence. Additionally, the `TAG_REGs` run-time registers could be used to track tag changes for heap allocations. Global variables could instead be treated as root level pointers and assigned a static tag.

Performance improvements. While our prototype outperforms many existing solutions in the literature while providing stronger security guarantees for the stack, it incurs moderate overhead in specific scenarios. This overhead mainly arises from memory tagging operations, as each function needs to tag and untag memory in its prologue and epilogue. Persistent memory tags, as proposed by StickyTags [101], could mitigate this drawback.

Additional memory tagging architectures. As discussed in Section 6.4, several memory tagging architectures have been proposed in the industry. While some niche architectures lack widespread adoption [158, 82], the RISC-V proposal [108] represents a promising step towards memory safety on an emerging CPU architecture. Although TAGShield's design and implementation are tailored for Arm MTE, we believe they could be adapted to support different tagging architectures. The principles underpinning our colouring schema and tag-integrity mechanisms are compatible with generic memory tagging architectures. Furthermore, our LLVM-based implementation, built on extensible components such as the LLVM Intermediate Representation (IR), could be readily adapted for other CPU architectures and tagging technologies.

Extended security evaluation. Section 6.6.1 analysed the security guarantees provided by TAGShield's colouring schema and tag-integrity mechanism. We demonstrated its effectiveness against two types of attackers to highlight the additional protection afforded by tag integrity. Furthermore, we demonstrated TAGShield effectiveness against the common attack vectors contained in the Juliet Test Suite [40]. However, we did not evaluate our solution against collections of CVEs or exploits targeting MTE protections. Currently, no collection of MTE-aware exploits exists to evaluate TAGShield against sophisticated attackers capable of forging tags. To address this, we plan to create a collection of exploits to serve as a foundation for testing advanced exploit mitigation techniques.

6.7 MTE in the literature

Memory safety has been the focus of much research in the past few decades due to the wide spread of memory vulnerabilities in unsafe languages such as C and C++. Numerous solutions have been proposed to address this issue, varying in their memory coverage, vulnerability scope, and impact on memory and run-time performance.

Classic solutions, such as memory sanitisers [226], play a significant role in detecting memory vulnerabilities at run-time. However, hardware-based alternatives have gained popularity for their lower performance impact. Among these, Arm has introduced two notable hardware extensions to mitigate memory vulnerabilities: Pointer Authentication Code (PAC) [86, 115, 157, 168] and Memory Tagging Extension (MTE) [217, 168, 101, 156, 197, 54]. Both have been utilised in research and commercial-off-the-shelf (COTS) products, with MTE emerging as the more performance-friendly option. While MTE effectively detects memory violations, the literature lacks robust yet efficient solutions capable of countering sophisticated attackers, such as those described in Section 6.3.1. This section compares TAGShield to similar MTE-based solutions, highlighting its advancements in memory protection.

Color My World [156] is one of the pioneering solutions to adopt MTE. Instead of utilising the 16 MTE tags as a probabilistic defence, Liljestrand et al. employ two tags to deterministically protect a subset of stack allocations. They classify allocations as either unsafe (vulnerable to exploitation) or safe (not vulnerable), tagging all safe allocations with a *safe tag* and the rest with an *unsafe tag*. This approach prevents unsafe allocations from corrupting safe ones, complemented by lightweight instrumentation to protect the unsafe tag. Although *Color My World* provides strong security guarantees with an acceptable overhead of 13.6%, it does not cover the entire stack, leaving unsafe allocations susceptible to exploitation.

In contrast, *StickyTags* [101] extends coverage to the entire stack, focusing on efficiently addressing spatial vulnerabilities like buffer overflows and underflows. Gorter et al. significantly reduce the number of memory tagging operations by tagging memory once and reusing the tags. Their approach involves dividing the stack and heap into zones and tagging memory in each zone with different colours. Instead of tagging each new allocation in the function preamble, objects are allocated within pre-tagged zones. While this approach achieves production-ready performance, it targets only a subset of vulnerabilities and remains vulnerable to tag forgery.

Similarly, *Zometag* [217] combines classic memory tagging with a zone-based approach to fully prevent buffer overflows. To enhance native MTE security guarantees, objects are grouped into *blue zones* of up to 14 items, alternating with inaccessible *red zones*. By preventing pointers from reaching beyond these red zones, Zometag ensures overflows are caught either by MTE checks or red zones. Despite its strong security guarantees, Zometag incurs high overhead and imposes source code constraints, particularly limiting certain types of pointer arithmetic.

Some works narrow their scope to specific scenarios rather than addressing general

memory vulnerabilities. *HAKC* [168] combines PAC and MTE to provide a compartmentalisation technique for Linux Kernel Modules (LKMs). Developers can annotate source code to define data-flow and control-flow dependencies between module subcomponents. HAKC processes these annotations to create an instrumented binary that uses PAC and MTE to prevent inter-component interference.

MTSan [54], on the other hand, introduces a binary sanitiser for run-time detection of memory errors with high precision and coverage. Leveraging MTE for spatial and temporal checks, MTSan improves performance compared to traditional software-based approaches. However, its high overhead renders it impractical for live systems.

A few works in the literature propose hardware modifications to boost MTE security guarantees and overcomes its 4-bit tags limitations. The first one is by Partap et. al [197] who propose a new hardware design that increases the size of each tag without incrementing the burden on tagged memory. Specifically, they propose a new technique to store tags in memory for big allocations: rather than storing the tag repeatedly for every 16 byte (tag granule), they design a page-level BTree structure to keep track of the size of the tagged allocations. This allows the use of bigger tags (less chances of collision) while reducing the memory overhead in most cases.

Multi-Tag [251] introduces hardware support for an additional page-level tag stored in the Page Table Entry (PTE). This approach utilises Arm's TBI feature to store the extra tag in the pointer alongside the MTE tag. Minimal hardware changes enable verification of page-level tags during each memory access to detect mismatches. Although this method promises strong security and performance, it requires hardware changes, limiting its practicality and compatibility with existing and upcoming devices.

6.8 Summary

This chapter introduced TAGShield, a novel approach to memory safety that fortifies the stack memory of vulnerable applications. TAGShield is an exploitation mitigation technique composed of two main components: a memory tagging schema, that ensures that corruptible pointers cannot be used freely to access arbitrary memory, and a tag integrity technique to make pointers' tags persistent and protected against advanced run-time attacks. In particular, we showed how the methodical application of tags in memory, combined with our tag integrity schema that prevents tags from being forged, can thwart powerful attackers. With our performance evaluation we showed that TAGShield incurs reasonable overhead in many scenarios, with several test cases reporting little to no performance degradation, especially in real-world applications.

This demonstrate TAGShield compatibility with production environments.

Chapter 7

Conclusions

Software and hardware tightly collaborate for the security of computer systems. Traditional security primitives, such as memory isolation, are enabled by a combination of hardware components and software layers, joined in co-designs to provide flexible and efficient security guarantees. Trusted Computing has emerged as a pivotal tool for enhancing the security of modern computer systems, offering a robust suite of security primitives that form the foundation for sophisticated secure systems. Trusted Computing traditionally relies on expensive hardware-based Roots of Trust (RoTs), such as Trusted Execution Environments (TEEs), and complex software layers to introduce a strongly isolated environment where trusted code and security services can run. Unfortunately, these complex solutions heavily rely on hardware that is only available in mid- to high-end devices because of its cost and complexity. Although some chip manufacturers are trying to port this hardware to lower-end devices — e.g. Arm TrustZone-M for the ARMv8-M architecture — a significant portion of low-end devices lack the resources to adopt any hardware security feature. These devices, which are extremely popular in the Internet of Things (IoT) and embedded systems, are often unprotected and left vulnerable to direct attacks, or exploited as pivot points for targeting other networked systems.

In traditional Trusted Computing architectures, software plays a crucial role by managing the underlying hardware security primitives and building comprehensive security services. On lower-end devices, however, software must go further, establishing the RoT from scratch with minimal hardware support. Creating cost-effective security architectures on these constrained systems is challenging, requiring a careful design to balance the flexibility and low performance of software-based security primitives. Software-hardware co-designs play a crucial role in this context, extending — or even

creating — basic security primitives for low-end devices. By leveraging the existing hardware, these co-designs can establish a cheaper and more robust security framework, while software enhances or extends these capabilities to reinforce the underlying hardware's security guarantees. Interestingly, software-hardware co-designs can also be used to enhance the security of high-end devices, by introducing additional security guarantees or mitigating specific limitations of the hardware.

This thesis explores the effectiveness of software-hardware co-designs by exploring the role of software in building Trusted Computing architectures and security services. By proposing both software-only Trusted Computing solutions and hybrid approaches combining hardware-assisted and software defences, we demonstrated that software can effectively bridge the security gap for low-end systems while enhancing the security guarantees of hardware-based techniques. Despite the technical challenges of implementing software-based RoTs, this work has showcased their adaptability highlighting their potential to unify security frameworks and extend protection to traditionally overlooked systems.

By advancing software-based security solutions and emphasizing the indispensable role of software in legacy and modern systems, this work paves the way for scalable, adaptable, and robust Trusted Computing frameworks across all device classes.

7.1 Summary of contributions

In this dissertation, we investigated the effectiveness and importance of software-hardware Trusted Computing architectures and services, advancing security for both low-end and high-end devices through innovative software-hardware co-designs. Our work highlights the critical role of software in enhancing system security, presenting a comprehensive software-based security stack for resource-constrained devices and a software-augmented exploit mitigation technique for high-performance systems.

To introduce the reader to the role that hardware and software play in ensuring security, **Chapter 2** provides an in-depth overview of a common low-end security hardware component: the Memory Protection Unit (MPU). This component, specific to embedded systems, is compared to its *larger sibling*, the Memory Management Unit (MMU), a critical feature in high-end systems. Beyond the MPU's limited security capabilities compared to the MMU, the contrasting programming paradigms of low-end and high-end systems present significant challenges for security. High-end systems are designed with security in mind, allowing application developers to rely on a trusted kernel and focus on functionality. In contrast, embedded system developers must often bear the

responsibility for security, including hardware configuration, firmware development, and managing various security components (when available). On these systems, software thus plays a crucial role and must be carefully crafted to fully leverage the hardware's security features.

Interestingly, the challenges of configuring the limited hardware introduce bugs, vulnerabilities, and misconfigurations, rendering low-end systems inherently insecure. Even the presence of Embedded Operative Systems (EOSs) often fails to mitigate these issues, as proper configuration remains the programmer's responsibility. To illustrate the severity of these challenges, in Chapter 2 we examine three examples across two architectures, MSP430 and ARMv7-M, demonstrating how MPU security can be bypassed due to incorrect configuration. Furthermore, we highlight how certain system features, such as software interrupts, can be exploited to circumvent MPU protection — even when a secure EOS is carefully designed — effectively showing how fragile security can be on low-end systems. To mitigate these issues, we provide guidelines for proper MPU configuration, underscoring the complexity of this task. This chapter established the fragility of hardware security components in embedded systems, demonstrating their deep dependence on proper software layers. Furthermore, we brought to light some of the intrinsic limitations of hardware to motivate the use of software-based security approaches, especially on devices that lack security hardware components.

Implementing software-based security, however, is not trivial. Security in any computer system must originate from a Root of Trust (RoT). While high-end systems often incorporate advanced hardware components such as TPMs and TEEs, the constrained hardware of low-end systems lacks these modules. As a result, security services like Remote Attestation or Secure Code Update cannot be readily deployed. To worsen matters, even basic security modules like the MPU are absent in certain device classes, making it exceedingly difficult to create any security layer. **Chapter 3** addresses this gap by introducing PISTIS, our Trusted Computing Module (TCM), which serves as the foundation for our security stack, a software-based TEE that enables the secure execution of Trusted Applications. In this chapter, we explore the challenges of achieving acceptable security guarantees in low-end systems, justifying the need for a software-based TEE through a detailed comparison with the state-of-the-art. We then describe the design and implementation of PISTIS, developed from the ground up without relying on pre-existing hardware modules or security features. Central to PISTIS is the core security primitive of memory isolation, which we demonstrate can be achieved entirely in software using a combination of software instrumentation, code verification, and run-time virtualisation. Specifically, PISTIS defines memory boundaries (i.e. an

Access Policy) and inspects the instructions of untrusted applications to ensure compliance. Non-compliant instructions are either approved or replaced with calls to *virtual functions* that enforce compliance at run-time. We detail the components of the TCM, including (i) a custom toolchain for code instrumentation, (ii) an onboard verifier for binary approval, and (iii) a run-time core for enforcing instruction compliance with the security policy. Additionally, we review the TEE's features, such as compatibility with interrupts and DMA operations, as well as security services like Remote Attestation and Secure Code Update. To highlight the potential of software-based approaches in the absence of hardware security, we formalise PISTIS's design and prove the correctness of its security primitives. PISTIS represents a significant step towards a software-based security stack for low-end devices, demonstrating the feasibility of implementing a TEE without relying on hardware security components.

Q1: What is the role of software-hardware co-designs in securing low-end devices? With these first two chapters we have shown that software-hardware co-designs are crucial to secure low-end devices. While hardware security components are often absent in these systems, software can be used to build security primitives that can effectively replace them. By leveraging the existing hardware, software can establish a cost-effective security framework that provides robust security guarantees.

Software-based approaches push the boundaries of software-hardware co-designs, filling security gaps that would otherwise remain unaddressed A key strength of these security techniques is their adaptability to diverse scenarios and environments. One particularly challenging environment is intermittent computing, where devices operate without a reliable power source. To manage potential power outages, applications often use checkpoints to save and restore their state. However, attackers can exploit these checkpoints, compromising system functionality. To address this issue and demonstrate the flexibility of our TCM, **Chapter 4** introduces MPI, a security extension that ensures secure checkpoint generation and restoration. MPI extends the TCM's functionality with accessible APIs for checkpoint generation. It enhances the TEE's security guarantees to protect the confidentiality and integrity of checkpoint images, thwarting attacks aimed at corruption or extraction. To handle power failures during checkpoint creation, MPI employs a double-buffer technique, maintaining the previous checkpoint as valid until a new one is successfully created. Additionally, we replace the original TCM bootloader with a trusted version that restores the appropriate checkpoint for both the TCM and the untrusted application whenever the device resumes operations.

Q2: Can software-based Trusted Computing provide a cost-effective and versatile RoT for low-end devices? PISTIS and MPI presented a software-based TCM that can effectively bring effective security guarantees to low-end devices. We showed that software can be integrated into existing systems inexpensively, and it can be adapted to multiple scenarios and contexts. By targeting very resource-constrained devices, we showed the flexibility and versability of software-based approaches.

With MPI, we demonstrated how our security stack adapts to various contexts and environments, meeting the security demands of intermittent computing through a secure checkpoint system. These first three chapters make a strong case for software-based security solutions, establishing a security stack resembling a traditional hardware-based TEE, such as TrustZone. Notably, software can be used to build other security services on top of the TCM. Although TEEs usually host Trusted Applications (TAs), i.e. typically passive security services operating only when requested (e.g. Remote Attestation), software-hardware Trusted Computing co-designs enable other types of services. While passive security service ensures controlled execution, this paradigm can limit the scope of available security services. To expand the capabilities of our TCM, **Chapter 5** introduces a novel active security service: a Control-Flow Integrity (CFI) scheme that can thwart run-time attacks from hijacking applications through stack manipulations.

Specifically, we address the issue of Return-Oriented Programming (ROP) attacks and evaluate existing hardware-based mitigations, presenting a novel software-based defence to thwart ROP attackers. FLASHADOW is a software-hardware co-design for a *shadow stack* that safeguards the integrity of function return addresses, ensuring they cannot be hijacked to alter application control flow. Our solution leverages the Flash memory and introduces a minimal software layer, built into the TCM, to protect the running applications. FLASHADOW actively monitors untrusted applications, storing return addresses in a secure memory area within the TCM. During return statements, control is transferred to FLASHADOW, which retrieves the correct return address from the shadow stack. FLASHADOW incorporates features into the `safe_call` and `safe_return` virtual functions of the TCM to manage the shadow stack's push and pop operations. In this chapter we proposed an implementation of our shadow stack based on Flash memory, showing how a double-pointer memory structure and carefully selected memory erasure operations are fundamental to safeguard the performance of this approach. This additional protection sensibly improves the security of the untrusted

part of the system, activivelly mitigating attacks aiming at disrupting its operations.

Q3: Can we build effective software-based security services on top of these devices? Although traditional Trusted Computing architectures support Trusted Applications for security services, achieving a similar security paradigm on low-end devices remains challenging. Despite the limited resources of our TCM, FLASHADOW demonstrated that it is possible to build active security services that effectively protect the system from run-time attacks. This reaffirms the versatility of software, which can deliver robust security services even in the absence of dedicated hardware primitives.

These four chapters have served for the building of a software-based security stack to fill the security gap between low- and high-end systems, while demonstrating the flexibility of software-hardware Trusted Computing co-designs. Notably, high-end systems still heavily rely on software to manage and harness the security primitives of the underlying hardware. However, while the highly constrained resources of low-end devices imply the need for a purely software-based Root of Trust (RoT), high-end systems can leverage more hybrid approaches where software builds on top of hardware security primitives. If, on the one hand, hardware is always coupled with a software component in charge of configuring and managing it (e.g. firmware or kernel), on the other hand software can go beyond that and boost the security guarantees of the hardware. This can be accomplished once again with software-hardware co-designs which, leveraging more sophisticated hardware, employ a more lightweight software layer. **Chapter 6** focuses on how this can be achieved in the context of memory tagging. This fairly recent hardware technique allows to link pointers to the memory they point to, ensuring that any misuse of pointers is either caught or prevented. Specifically, it first allows to assign tags, or colours, to both memory and pointers; second, it triggers an hardware check upon memory access, ensuring that the tag of the pointer matches that of the memory. We proposed TAGShield, a memory tagging schema that uses software to address one of the intrinsic limitations of memory tagging: the integrity of the tags themselves. We show how tags can be made persistent at run-time through a lightweight software layer. First, TAGShield uses CPU registers to keep track of the authentic tags at run-time. Second, it makes tags persistent by restoring their authentic value before pointers are dereferenced, effectively augmenting the security guarantees of the memory tagging architecture. If, on the one hand, this lowers the efficiency of memory tagging, increasing the impact on run-time overhead, on the other hand, it introduces extra security guarantees.

Q4: Can software-hardware co-designs further boost the security guarantees provided by hardware primitives on high-end systems? Software-based approaches truly shine on low-end devices. However, software can also enhance existing hardware security primitives. TAGShield demonstrates this by showing how a novel hardware primitive — memory tagging — can be extended and improved with a lightweight software layer. Interestingly, unlike previous software-hardware co-designs, TAGShield takes a more hardware-centric approach while remaining tightly integrated with software.

7.2 A look at the bigger picture

In the previous chapters, we examined various components of our security stack and demonstrated their applicability in protecting low-end devices through software-based techniques. While we believe these contributions significantly advance the security community by proposing novel techniques and addressing previously unresolved challenges, it is crucial to contextualise our work within the broader scope of security research.

Previously, we emphasised how software-based approaches can address gaps created by the absence of security-related hardware components, such as TEEs, MMUs or even MPUs. However, this is just one of the many advantages that software-based security offers. One particularly important benefit lies in the scalability and flexibility of software compared to hardware. For example, consider the TEE technologies currently available on COTS devices. Each major chip manufacturer has proposed its own version of TEE. Although their fundamental purpose — to provide a secure and isolated execution environment that resists interference — remains consistent, their implementations differ significantly. Two prominent examples illustrate this disparity: Intel SGX [62] and Arm TrustZone [22]. While both enable secure software execution, they achieve so with very different approaches. Intel SGX is a technology that supports user-level application enclaves: every application can implements its own secure environment where to execute critical code. These enclaves are hardware-protected and can be accessed only through specific endpoints. In contrast, TrustZone separates the SoC in two: the Rich Execution Environment (REE) and the secure world (TEE). While the former contains traditional hypervisor-kernel-applications systems, the secure world can host a complex OS to manage system-level security services in the forms of Trusted Applications (TAs). The secure world comes with additional privilege levels, dedicated hardware components and a complex API system to handle all the REE-TEE interactions.

Given these very different technologies, one operating at user-level the other one at system-level, even a standard security service, such as a cryptographic engine or remote attestation, requires vastly different implementations across these platforms. This lack of uniformity complicates the interoperability of security services and, ultimately, hinders the widespread adoption of Trusted Computing solutions.

In contrast, software-based approaches inherently avoid this drawback. By deploying our security stack across different architectures, albeit with some necessary modifications to accommodate architectural specifics, we can retain the same core logic and avoid substantial redesigns of security services. This characteristic represents a key advantage of software-based security, especially as the proliferation of devices increases and the demand for robust security becomes ever more pressing.

We argue that this flexibility and adaptability will be paramount in the future of security solutions, with software playing a bigger role on hardware-rich devices. The capability to standardise security services across diverse hardware platforms not only reduces development effort but also fosters greater adoption of security practices in both existing and emerging systems. Interestingly, reducing the hardware reliance with carefully crafted software layers can go a long way towards scalable performance-friendly security solutions. One important showcase of the potential of software-based security components across diversified computer systems is given by the CrossCon Project.

7.2.1 The CrossCon project

To validate our thesis, we analyse the impact of our contribution within the CrossCon project [66]. CrossCon (Cross-platform Open Security Stack for Connected Devices) is a european initiative focused on developing a cross-platform security stack to enable the interoperability of security services, thereby promoting the adoption of defence techniques across a wide range of devices. A significant portion of this thesis was conceived as a core component of this project, serving as a bridge between the higher-end devices and the low-end embedded systems.

Specifically, PISTIS was developed as the foundational low-level building block for enabling the CrossCon stack on the most resource-constrained devices. With PISTIS, we demonstrated that devices of various classes can securely communicate and rely on one another, all within a security context where each device can be protected with nearly equal guarantees.

Notably, within the CrossCon project, we successfully ported PISTIS to more capable low-end devices equipped with ARMv7-M processors and an MPU. This enhanced

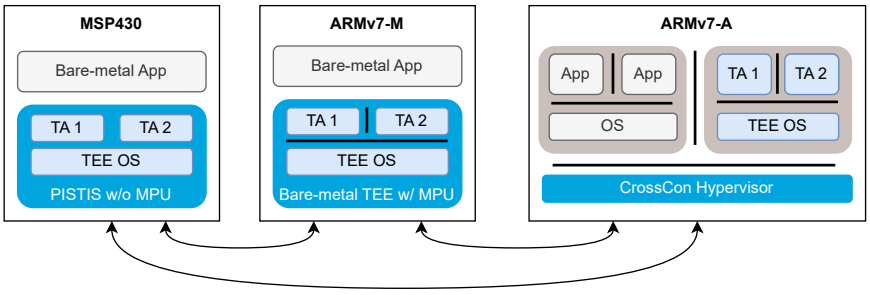


Figure 7.1: An overview of the CrossCon project, which aims at creating a cross-platform security stack to allow the inter-operability of security services. Three different architectures are depicted, each equipped with a different TEE technology powered by software, all three providing similar security guarantees and interoperability of their services.

version [65] of our software-based bare-metal TEE achieves near-complete compliance with the GlobalPlatform standard [94], enabling full interoperability of Trusted Applications (TAs). This standard defines a common set of APIs that each Trusted OS and TA should implement to be easily ported among TEEs supporting the same standard. This PISTIS port underscores the potential of software-based defences to provide a consistent set of security guarantees across devices of different classes.

Figure 7.1 provides an overview of the CrossCon project.

7.2.2 Limitations of software-based approaches

The software-based approaches explored in this thesis offer numerous advantages to security. While the benefits of software have been thoroughly discussed in previous sections and chapters, it is equally important to highlight and understand the limitations of these techniques. The first significant limitation lies in the run-time performance. Software is versatile and can be used to replace, or even emulate, many hardware components, such as the Memory Management Unit (MMU) and Memory Protection Unit (MPU). However, operations performed in software are inherently slower than their hardware counterparts. This disparity becomes more pronounced when dealing with complex components like Trusted Execution Environments (TEEs). As a result, achieving an optimal balance between software and hardware is crucial in co-design efforts. Despite its performance drawbacks, software may still be the only practical solution in certain scenarios, making it an indispensable tool in the security landscape.

Another limitation of software lies in its weaker physical security guarantees. For example, when comparing a traditional hardware-based TEE with the security stack

proposed in this thesis, the latter is significantly more vulnerable to physical attacks. An attacker with access to a device running PISTIS could use one of its interfaces (e.g. JTAG) to erase the entire Trusted Computing Base (TCB) and completely compromise the device's defences. While orthogonal defences exist to mitigate such threats [209], hardware TEEs often come equipped with robust, integrated protections designed to thwart physical attacks. It is worth noting, however, that the impact of a physical attacker is generally more localized compared to that of a remote adversary capable of compromising multiple devices within a network.

Finally, software is generally more susceptible to bugs than hardware. Bugs in security-critical software components can lead to catastrophic consequences, underscoring the importance of rigorous formal verification and thorough validation for deployment-ready techniques. By ensuring these processes are in place, the risks associated with software vulnerabilities can be significantly mitigated.

7.3 Future work directions

The work presented in this thesis has explored critical challenges in computer systems security, with a particular focus on the role of software in constructing robust security defenses. While each chapter has contributed significantly to the field, numerous avenues remain open for further research to strengthen the security guarantees of the proposed defenses. Additionally, several challenges must be addressed to enhance the effectiveness and efficiency of software-based security mechanisms.

One promising direction for future work is a comprehensive study of the current IoT security landscape. In particular, analyzing real-world firmware could provide valuable insights into the actual security needs of low-end devices. While numerous attacks and vulnerabilities have been documented, a broader analysis is necessary to determine the prevalence of these issues. This study would involve collecting and analyzing Common Vulnerabilities and Exposures (CVEs) as well as both open- and closed-source firmware to assess the security landscape of such devices.

Another key area of research is evaluating the real-world performance impact of software-based Roots of Trust (RoT). While this thesis has assessed their performance costs, an empirical study is needed to determine how much of this overhead is acceptable in practice. This could involve surveying the performance requirements of security-critical IoT applications and assessing the trade-offs imposed by software-based RoTs. The energy sector, which increasingly relies on IoT devices for monitoring and control, presents an ideal case study for such an evaluation.

Beyond these studies, further refinement of the proposed solutions could enhance both their performance and effectiveness. The security services built on top of our Trusted Computing Module (TCM) could be expanded further. For example, the Control Flow Integrity (CFI) mechanisms of FLASHADOW could be improved with more fine-grained forward-edge protection. Additionally, other security services traditionally reliant on hardware, such as secure boot, could be explored within the context of our TCM. Our work with Arm MTE could also be extended to cover additional memory types, such as the heap, to provide more comprehensive security protections. As memory tagging architectures continue to evolve, porting TAGShield to emerging standards could offer valuable insights into the generalizability and scalability of our approach.

The adaptability of our software-based TCM to additional architectures, such as ARMv6-M and RISC-V, is another critical research avenue. RISC-V, in particular, is an emerging architecture that lacks a built-in Trusted Computing framework, making it a strong candidate for these security techniques. Evaluating the cost of porting our solutions to new architectures would provide concrete evidence of the scalability and flexibility of our security stack.

New computing platforms continue to emerge with varying hardware capabilities. For instance, Cortex-R processors lack TrustZone-M, requiring alternative security solutions that leverage available hardware features. Investigating how our TCM design would need to be adapted for such architectures would further demonstrate its generalizability. Similarly, high-end devices are incorporating increasingly advanced security features that may necessitate software enhancements similar to those introduced by TAGShield. The new Realm Management Extension (RME) in Armv9, for instance, presents opportunities for novel software-based security defenses that could further enhance system security.

Another emerging research direction is the security of Graphics Processing Units (GPUs). As GPUs are increasingly used for general-purpose computing, they present new attack surfaces that must be addressed. Investigating how our software-based security mechanisms can be adapted to GPUs would be crucial in ensuring their robustness. This includes analyzing vulnerabilities specific to GPU architectures, exploring memory isolation techniques, and assessing the feasibility of extending software-based Roots of Trust to these processing units. Given the rise of heterogeneous computing environments, integrating GPUs into our security model could enhance overall system security and provide a broader defense framework.

In conclusion, software-hardware co-design remains central to modern Trusted Computing architectures. There is significant potential for further research demonstrating

how software, when properly designed, can greatly enhance system security across a wide range of hardware platforms.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] Martín Abadi and Gordon D Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security (TISSEC)*, 15(2):1–29, 2012.

[3] AbdElaziz Saad AbdElaziz AbdElaal, Kai Lehniger, and Peter Langendörfer. Incremental code updates exploitation as a basis for return oriented programming attacks on resource-constrained devices. In *2021 5Th cyber security in networking conference (CSNet)*, pages 55–62. IEEE, 2021.

[4] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[5] Nada Abughazaleh, R Bin, and Mai Btish. Dos attacks in iot systems and proposed solutions. *Int. J. Comput. Appl*, 176(33):16–19, 2020.

[6] Inc. Advanced Micro Devices. AMD Secure Encrypted Virtualization (SEV). `https://www.amd.com/en/developer/sev.html`, 2023. Last accessed: Feb. 23, 2025.

[7] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 70–81, 2019.

[8] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, volume 10, page 96, 2009.

[9] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel trust domain extensions (tdx) security review. Technical report, Google technical report, 2023.

[10] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. μrai: Securing embedded systems with return address integrity. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[11] Naif Saleh Almakhdhub, Abraham A Clements, Saurabh Bagchi, and Mathias Payer. μrai: Securing embedded systems with return address integrity. In *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[12] Mahmoud Ammar and Bruno Crispo. Verify&Revive: Secure Detection and Recovery of Compromised Low-end Embedded Devices. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[13] Mahmoud Ammar and Bruno Crispo. Verify&revive: Secure detection and recovery of compromised low-end embedded devices. In *Annual Computer Security Applications Conference*, pages 717–732, 2020.

[14] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. SμV—The Security MicroVisor: A Formally-verified Software-based Security Architecture for the Internet of Things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901, 2019.

[15] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. Sμv—the security microvisor: A formally-verified software-based security architecture for the internet of things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901, 2019.

[16] Mahmoud Ammar, Bruno Crispo, and Gene 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.

[17] Mahmoud Ammar, Wilfried Daniels, Bruno Crispo, and Danny Hughes. SPEED: Secure Provable Erasure for Class-1 IoT Devices. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 111–118, 2018.

[18] James P Anderson et al. Computer security technology planning study. Technical report, Citeseer, 1972.

[19] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[20] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai botnet. In *26th USENIX security symposium*, pages 1093–1110, 2017.

[21] Emekcan Aras, Mahmoud Ammar, Fan Yang, Wouter Joosen, and Danny Hughes. Microvault: Reliable storage unit for iot devices. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 132–140. IEEE, 2020.

[22] ARM. TrustZone technology for Armv8-A. `https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-a\#armv8-a`, 2011. Last accessed: Feb. 23, 2025.

[23] ARM. TrustZone technology for Armv8-M. `https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m`, 2015. Last accessed: Feb. 23, 2025.

[24] ARM. TrustZone technology for the ARMv8-M architecture Version 2.0. `https://developer.arm.com/documentation/100690/0200/ARM-TrustZone-technology`, 2022. Last accessed: Feb. 23, 2025.

[25] ARM. Learn the Architecture - Providing Protection for Complex Software. `https://developer.arm.com/documentation/102433/0100`, 2023. Last accessed: Feb. 23, 2025.

[26] ARM ARM. Architecture reference manual-armv8, for armv8-a architecture profile. *ARM Limited, Dec*, 2017.

[27] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security Framework for the Analysis and Design of Software Attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1–12, 2013.

[28] Hafiz Areeb Asad, Erik Henricus Wouters, Naveed Anwar Bhatti, Luca Mottola, and Thiemo Voigt. On securing persistent state in intermittent computing. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, pages 8–14, 2020.

[29] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[30] Authors. Github repository.

[31] Zelalem Birhanu Aweke and Todd Austin. usfi: Ultra-lightweight software fault isolation for iot-class devices. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1015–1020. IEEE, 2018.

[32] Matthew Ball. It spending to accelerate 6

[33] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016.

[34] Ayush Bansal and Debadatta Mishra. A practical analysis of rop attacks. *arXiv preprint arXiv:2111.03537*, 2021.

[35] Markus Bauer and Christian Rossow. Cali: Compiler assisted library isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS'21)*. Association for Computing Machinery, 2021.

[36] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, volume 10, page 1, 2005.

[37] Naveed Anwar Bhatti and Luca Mottola. Harvos: Efficient code instrumentation for transiently-powered embedded sensing. In *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 209–220. IEEE, 2017.

[38] Lorenzo Binosi, Gregorio Barzasi, Michele Carminati, Stefano Zanero, and Mario Polino. The illusion of randomness: An empirical analysis of address space layout randomization implementations. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1360–1374, 2024.

[39] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[40] Tim Boland and Paul E Black. Juliet 1. 1 c/c++ and java test suite. *Computer*, 45(10):88–90, 2012.

[41] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of the 52nd Annual Design Automation Conference*, page 6, New York, New York, USA, jun 2015. ACM.

[42] Michael Buettner, Ben Greenstein, and David Wetherall. Dewdrop: an energy-aware run-time for computational rfid. In *Proc. USENIX NSDI*, pages 197–210, 2011.

[43] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[44] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

[45] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 388(1-29):3, 2016.

[46] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409, 2009.

[47] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[48] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[49] David R Chase, Mark Wegman, and F Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, 1990.

[50] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[51] Qian Chen and Robert A Bridges. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *2017 16th IEEE International Conference on machine learning and applications (ICMLA)*, pages 454–460. IEEE, 2017.

[52] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX security symposium*, volume 5, page 146, 2005.

[53] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[54] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. Mtsan: A feasible and practical memory sanitizer for fuzzing cots binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 841–858, 2023.

[55] Yu-Chang Chen and Shih-Wei Li. Hemate: Enhancing heap security through isolating primitive types with arm memory tagging extension. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–11, 2024.

[56] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N Asokan, and Danfeng Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–36, 2021.

[57] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2012.

[58] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 65–82, 2018.

[59] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting Bare-metal Embedded Systems with Privilege Overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303. IEEE, 2017.

[60] Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 514–530, 2016.

[61] European Commission. Cyber Resilience Act. `https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act`, 2024. Last accessed: Feb. 23, 2025.

[62] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[63] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, 2014.

[64] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[65] CrossCon. Bare metal TEE with MPU. `https://github.com/crosscon/baremetal-tee/tree/main/MPU-version`, 2024. Last accessed: Feb. 23, 2025.

[66] Crosscon. Crosscon. `https://crosscon.eu/dissemination-material/crosscon-white-paper/`, 2024. Last accessed: Feb. 23, 2025.

[67] CWE. 2024 CWE Top 25 Most Dangerous Software Weaknesses, 2024.

[68] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[69] Daniel Danner, Rainer Müller, Wolfgang Schröder-Preikschat, Wanja Hofer, and Daniel Lohmann. Safer sloth: Efficient, hardware-tailored memory protection. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 37–48. IEEE, 2014.

[70] Dave Muoio. DA warns of Bluetooth Low Energy vulnerability affecting connected medical devices. `https://www.mobihealthnews.com/news/fda-warns-bluetooth-low-energy-vulnerability-affecting-connected-medical-devices`, 2020. Last accessed: Feb. 23, 2025.

[71] John H Davies. *MSP430 microcontroller basics*. Elsevier, 2008.

[72] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. Fixer: Flow integrity extensions for embedded risc-v. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 348–353. IEEE, 2019.

[73] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–954, 2020.

[74] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. Camaroptera: A long-range image sensor with local inference for remote sensing applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 2022.

[75] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[76] Daniel Dinu, Archanaa S Khrishnan, and Patrick Schaumont. Sia: secure intermittent architecture for off-the-shelf resource-constrained microcontrollers. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 208–217. IEEE, 2019.

[77] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J Walls, and John Criswell. Holistic control-flow protection on real-time embedded systems with kage. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2281–2298, 2022.

[78] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[79] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[80] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, volume 17, pages 1–15, 2017.

[81] Richard Earnshaw. Glibc mte support.

[82] Inc. Eklektix. Tagged memory and minion cores in the lowrisc soc.

[83] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[84] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *Ndss*, volume 12, pages 1–15, 2012.

[85] Maik Ender, Amir Moradi, and Christof Paar. The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs. In *29th USENIX Security Symposium*, pages 1803–1819, 2020.

[86] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.

[87] Federal Trade Commission. Equifax Data Breach Settlement . `https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement`, 2024. Last accessed: Feb. 23, 2025.

[88] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the penglai enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294, 2021.

[89] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. A Minimalist Approach to Remote Attestation. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[90] Anmin Fu, Weijia Ding, Boyu Kuang, Qianmu Li, Willy Susilo, and Yuqing Zhang. Fh-cfi: Fine-grained hardware-assisted control flow integrity for arm-based iot devices. *Computers & Security*, 116:102666, 2022.

[91] Inc. Gentoo Foundation and Förderverein Gentoo e.V. Gentoo linux.

[92] Rahul George, Mingming Chen, Kaiming Huang, Zhiyun Qian, Thomas La Porta, and Trent Jaeger. Optisan: Using multiple spatial error defenses to optimize stack memory protection within a budget. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7195–7212, 2024.

[93] Zahra Ghodsi, Siddharth Garg, and Ramesh Karri. Optimal checkpointing for secure intermittently-powered iot devices. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 376–383. IEEE, 2017.

[94] Global Platform. TEE Internal Core API Specification. `https://globalplatform.org/specs-library/tee-internal-core-api-specification/`, 2021. Last accessed: Feb. 23, 2025.

[95] GlobalPlatform. Introduction to trusted execution environments. `https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf`, 2018. Last accessed: Feb. 23, 2025.

[96] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213, 2019.

[97] Shyamnath Gollakota, Matthew S Reynolds, Joshua R Smith, and David J Wetherall. The emergence of rf-powered computing. *Computer*, 47(1):32–39, 2013.

[98] Google. Retrofitting Spatial Safety to Hundreds of Millions of Lines of C++, 2024.

[99] Google. Retrofitting spatial safety to hundreds of millions of lines of C++. `https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html`, 2024.

[100] Maria Gorlatova, John Sarik, Guy Grebla, Mina Cong, Ioannis Kymissis, and Gil Zussman. Movers and shakers: Kinetic energy harvesting for the internet of things. *IEEE Journal on Selected Areas in Communications*, 33(8):1624–1639, 2015.

[101] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky tags: Efficient and deterministic spatial memory error mitigation using persistent memory tags. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 217–217. IEEE Computer Society, 2024.

[102] Michele Grisafi, Mahmoud Ammar, and Bruno Crispo. On the (in) security of memory protection units: A cautionary note. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 157–162. IEEE, 2022.

[103] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. Pistis: Trusted computing architecture for low-end embedded systems. In *Proc. of USENIX Security*, 2022.

[104] Grisafi, Michele and Ammar, Mahmoud and Crispo, Bruno. MPU demos. `https://github.com/CybersecurityUnitn/MPU`. Last accessed: Feb. 23, 2025.

[105] Grisafi, Michele and Ammar, Mahmoud and Roveri, Marco and Crispo, Bruno. PISTIS Source Code . `https://github.com/CybersecurityUnitn/PISTIS`. Last accessed: Feb. 23, 2025.

[106] Grisafi, Michele and Ammar, Mahmoud and Roveri, Marco and Crispo, Bruno. FLASHadow Source Code. `https://github.com/MicheleGrisafi/Flashadow`, 2023. Last accessed: Feb. 23, 2025.

[107] Grisafi, Michele and Ammar, Mahmoud and Yildirim, Kasim Sinan and Crispo, Bruno. Source code of MPI. `https://github.com/CybersecurityUnitn/MPI`, September 2022. Last accessed: Feb. 23, 2025.

[108] RISC-V Main Group. Tech memory tagging - risc v working group.

[109] Philipp Gutruf, Vaishnavi Krishnamurthi, Abraham Vázquez-Guardado, Zhaoqian Xie, Anthony Banks, Chun-Ju Su, Yeshou Xu, Chad R Haney, Emily A Waters, Irawati Kandela, et al. Fully implantable optoelectronic systems for battery-free, multimodal operation in neuroscience research. *Nature Electronics*, 1(12):652–660, 2018.

[110] Andreas Hager-Clukas and Konrad Hohentanner. Dmti: Accelerating memory error detection in precompiled c/c++ binaries with arm memory tagging extension. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1173–1185, 2024.

[111] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, 2005.

[112] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application Memory Isolation on Ultra-low-power MCUs. In *USENIX Annual Technical Conference (ATC)*, pages 127–132, 2018.

[113] Josiah Hester and Jacob Sorber. Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–13, 2017.

[114] Josiah Hester, Kevin Storer, and Jacob Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–13, 2017.

[115] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. Cryptsan: Leveraging arm pointer authentication for memory safety in c/c++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 1530–1539, 2023.

[116] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015.

[117] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[118] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In *NDSS*, 2022.

[119] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Top of the heap: Efficient memory error protection of safe heap objects. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1330–1344, 2024.

[120] Kaiming Huang, Jack Sampson, and Trent Jaeger. Assessing the impact of efficiently protecting ten million stack objects from memory errors comprehensively. In *2023 IEEE Secure Development Conference (SecDev)*, pages 67–74. IEEE, 2023.

[121] imec. The medical implants of the future: faster, smarter and more connected. `https://www.imec-int.com/en/imec-magazine/imec-magazine-april-2020/the-medical-implants-of-the-future-faster-smarter-and-more-connected`, 2020. Last accessed: Feb. 23, 2025.

[122] Prof. M. U. Inamdar and Prof. M. S. Biradar. Human health monitoring system in abnormal condition using msp 430 to remote pc. *IJIERT - International Journal of Innovations in Engineering Research and Technology*, 2(4), 2015.

[123] Texas Instruments. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide . `https://www.ti.com/lit/ug/slau367p/slau367p.pdf`, 2020. Last accessed: Feb. 23, 2025.

[124] Texas Instruments. EnergyTrace Technology. `https://www.ti.com/tool/ENERGYTRACE`, 2021. Last accessed: Feb. 23, 2025.

[125] Internet Engineering Task Force (IETF). Terminology for Constrained-Node Networks. `https://datatracker.ietf.org/doc/html/rfc7228\#section-2.1`, 2014. Last accessed: Feb. 23, 2025.

[126] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882, 2018.

[127] Harshvardhan P Joshi, Aravindhan Dhanasekaran, and Rudra Dutta. Trading off a vulnerability: does software obfuscation increase the risk of rop attacks. *Journal of Cyber Security and Mobility*, pages 305–324, 2015.

[128] Jiawen Kang, Rong Yu, Sabita Maharjan, Yan Zhang, Xumin Huang, Shengli Xie, Hanna Bogucka, and Stein Gjessing. Toward secure energy harvesting cooperative networks. *IEEE Communications Magazine*, 53(8):114–121, 2015.

[129] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[130] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. *ACM SIGARCH Computer Architecture News*, 40(3):94–105, 2012.

[131] The Linux Kernel. Ksan.

[132] The Linux Kernel. Control-flow Enforcement Technology (CET) Shadow Stack, 2024.

[133] Rozan Khader and Derar Eleyan. Survey of dos/ddos attacks in iot. *Sustainable Engineering and Innovation*, 3(1):23–28, 2021.

[134] Arslan Khan, Dongyan Xu, and Dave Jing Tian. Ec: Embedded systems compartmentalization via intra-kernel isolation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2990–3007. IEEE, 2023.

[135] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.

[136] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. Tiktag: Breaking arm's memory tagging extension with speculative execution. *arXiv preprint arXiv:2406.08719*, 2024.

[137] Neil Klingensmith and Suman Banerjee. Hermes: A real time hypervisor for mobile and iot systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, pages 101–106, 2018.

[138] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[139] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the 9th European Conference on Computer Systems*, page 14, New York, NY, USA, 2014. ACM.

[140] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[141] Tim Kornau. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[142] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. Time-sensitive intermittent computing meets legacy

software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–99, 2020.

[143] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[144] Mohit Kumar. Ransomware Hijacks Hotel Smart Keys to Lock Guests Out of their Rooms. `https://thehackernews.com/2017/01/ransomware-hotel-smart-lock.html`, 2017. Last accessed: Feb. 23, 2025.

[145] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: Software-based Memory Protection for Sensor Nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349, 2007.

[146] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: software-based memory protection for sensor nodes. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 340–349, 2007.

[147] V. R. Kumari, P. Naga Lakshmi, V. Baby Soujanya, and R. Geetha. Msp 430 lunch box based smart irrigation system. *International Research Journal of Innovations in Engineering and Technology*, 5(4):79–82, 04 2021.

[148] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[149] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[150] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[151] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent Byunghoon Kang. Savior: Thwarting stack-based memory safety violations by randomizing stack layout. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2559–2575, 2021.

[152] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. Zipper stack: Shadow stacks without shadow. In *European Symposium on Research in Computer Security*, pages 338–358. Springer, 2020.

[153] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, pages 16–29, Berlin, Heidelberg, 2010. Springer-Verlag.

[154] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1901–1915, 2022.

[155] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[156] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N. Asokan. Color my world: Deterministic tagging for memory safety, 2022.

[157] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[158] Inc. Linux Kernel Organization. Sparc adi.

[159] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[160] Qi Liu, Kaibin Bao, and Veit Hagenmeyer. Binary exploitation in industrial control systems: Past, present and future. *IEEE Access*, 10:48242–48273, 2022.

[161] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 50(6):575–585, 2015.

[162] Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[163] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, 2018.

[164] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Transactions on Sensor Networks (TOSN)*, 16(1):1–24, 2020.

[165] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - specification*. Springer, 1992.

[166] Robert Margolies, Guy Grebla, Tingjun Chen, Dan Rubenstein, and Gil Zussman. Panda: Neighbor discovery on a power harvesting budget. *IEEE Journal on Selected Areas in Communications*, 34(12):3606–3619, 2016.

[167] Uwe F. Mayer. Linux/unix nbench.

[168] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakcs. In *NDSS*, pages 1–17, 2022.

[169] Francesca Meneghello, Matteo Calore, Daniel Zucchetto, Michele Polese, and Andrea Zanella. IoT: Internet of threats? A survey of practical security vulnerabilities in real IoT devices. *IEEE Internet of Things Journal*, 6(5):8182–8201, 2019.

[170] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. D-box: Dma-enabled compartmentalization for embedded applications. *arXiv preprint arXiv:2201.05199*, 2022.

[171] Microsoft. What is data execution prevention (dep)?

[172] Microsoft. Control Flow Guard for platform security. `https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard`, 2022. Last accessed: Feb. 23, 2025.

[173] Ingo Molnar. Exec shield. *http://lkml. org/lkml/2003/5/2/96*, 2003.

[174] Brendan Moran, Milosch Meriac, Hannes Tschofenig, and David Brown. A Firmware Update Architecture for Internet of Things Devices. *Internet Engineering Task Force, Internet-Draft*, 2019.

[175] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. A firmware update architecture for internet of things. *draft-ietf-suit-architecture-08*, 2021.

[176] Antonio Muñoz, Ruben Rios, Rodrigo Román, and Javier López. A survey on the (in) security of trusted execution environments. *Computers & Security*, 129:103180, 2023.

[177] Yeoul Na. -fbounds-safety. enforcing bounds safety for production c code. *EuroLLVM Developers' Meeting*, May 2023.

[178] Saman Naderiparizi, Aaron N Parks, Zerina Kapetanovic, Benjamin Ransford, and Joshua R Smith. Wispcam: A battery-free rfid camera. In *2015 IEEE International Conference on RFID (RFID)*, pages 166–173. IEEE, 2015.

[179] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[180] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.

[181] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. Camaroptera: A batteryless long-range remote visual sensing system. In *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*, pages 8–14, 2019.

[182] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[183] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank

Piessens. SANCUS: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium*, pages 479–498, 2013.

[184] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, 2013.

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

[186] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 641–646. IEEE, 2021.

[187] NXP. Edgelock® secure enclave, 2025.

[188] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 259–284. Springer, 2017.

[189] US Department of Security. Supply Chain Cybersecurity Principles. `https://www.energy.gov/sites/default/files/2024-06/DOE%20Supply%20Chain%20Cyber%20Princples%20June%202024.pdf`, 2024. Last accessed: Feb. 23, 2025.

[190] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. utango: an open-source tee for iot devices. *IEEE Access*, 10:23913–23930, 2022.

[191] Aleph One. Smashing the stack for fun and profit. phrack 7, 49 (november 1996), 1996.

[192] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[193] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. Softbound+ cets revisited: More than a decade later. In *Proceedings of the 17th European Workshop on Systems Security*, pages 22–28, 2024.

[194] Runyu Pan and Gabriel Parmer. Mxu: Towards predictable, flexible, and efficient memory access control for the secure iot. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–20, 2019.

[195] Runyu Pan and Gabriel Parmer. Sbis: Application access to safe, baremetal interrupt latencies. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 82–94. IEEE, 2022.

[196] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. Predictable virtualization on memory protection unit-based microcontrollers. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 62–74. IEEE, 2018.

[197] Aditi Partap and Dan Boneh. Memory tagging: A memory efficient design. *arXiv preprint arXiv:2209.00307*, 2022.

[198] Phoronix. Intel MPX Support Is Dead With Linux 5.6. `https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Is-Dead`, 2020. Last accessed: Feb. 23, 2025.

[199] Sandro Pinto and Cesare Garlati. Multi zone security for arm cortex-m devices. In *Embedded World Conference*, volume 2020, 2020.

[200] Sanndro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. Virtualization on trustzone-enabled microcontrollers? voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304. IEEE, 2019.

[201] LLVM Project. Llvm 18.1.8 release notes.

[202] LLVM Project. Llvm language reference manual.

[203] LLVM Project. Memtagsanitizer.

[204] LLVM Project. Scudo hardened allocator.

[205] LLVM Project. Stack safety analysis - llvm 19.0.0.

[206] Amir Rahmati, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P Burleson, and Kevin Fu. Tardis: Time and remanence decay in sram to implement secure protocols on embedded devices without clocks. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 221–236, 2012.

[207] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Codarr: Continuous data space randomization against data-only attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 494–505, 2020.

[208] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on rfid-scale devices. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 159–170, 2011.

[209] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper Resistance Mechanisms for Secure Embedded Systems. In *17th International Conference on VLSI Design. Proceedings.*, pages 605–611. IEEE, 2004.

[210] Reuters. Yahoo shares fall after latest security breach. `https://www.reuters.com/article/business/yahoo-shares-fall-after-latest-security-breach-idUSKBN144238/`, 2016. Last accessed: Feb. 23, 2025.

[211] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[212] Arman Roohi, Ronald F DeMara, Longfei Wang, and Selçuk Köse. Secure intermittent-robust computation for energy harvesting device security and outage resilience. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1–5. IEEE, 2017.

[213] Ravi Sahita, Vedvyas Shanbhogue, Andrew Bresticker, Atul Khare, Atish Patra, Samuel Ortiz, Dylan Reid, and Rajnesh Kanwal. Cove: Towards confidential computing on risc-v platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 315–321, 2023.

[214] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement*, 57(11):2608–2615, 2008.

[215] Martin Schönstedt, Ferdinand Brasser, Patrick Jauernig, Emmanuel Stapf, and Ahmad-Reza Sadeghi. Safetee: combining safety and security on arm-based microcontrollers. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 520–525. IEEE, 2022.

[216] Sean Michael Kerner, TechTarget. Crowdstrike outage explained: What caused it and what's next. `https://www.techtarget.com/whatis/feature/Explaining-the-largest-IT-outage-in-history-and-whats-next`, 2024. Last accessed: Feb. 23, 2025.

[217] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. Zometag: Zone-based memory tagging for fast, deterministic detection of spatial memory violations on arm. *IEEE Transactions on Information Forensics and Security*, 2023.

[218] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[219] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[220] A Seshadri, A Perrig, L van Doorn, and P Khosla. SWATT: Software-based Attestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 272–282. IEEE, may 2004.

[221] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *Distributed Computing in Sensor Systems*, pages 372–385, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[222] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, 2006.

[223] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[224] Muhammad Ali Siddiqi, Angeliki-Agathi Tsintzira, Georgios Digkas, Miltiadis G Siavvas, and Christos Strydis. Adding Security to Implantable Medical Devices: Can We Afford It? In *EWSN*, pages 67–78, 2021.

[225] Miguel Silva, David Cerdeira, Sandro Pinto, and Tiago Gomes. Operating systems for internet of things low-end devices: Analysis and benchmarking. *IEEE Internet of Things Journal*, 6(6):10375–10383, 2019.

[226] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[227] System Performance Evaluation Cooperative (SPEC). Spec cpu® 2006.

[228] T SPERRY. 386 VS 030-THE CROWDED FAST LANE. *DR DOBBS JOURNAL*, 13(1):16, 1988.

[229] V Stafford. Zero trust architecture. *NIST special publication*, 800:207, 2020.

[230] Geetha Devi Appari Surya Prasada Rao Borra1. A smart home security locker using microcontroller. *IOSR Journal of Engineering (IOSRJEN)*, 11:01–04, July 2021.

[231] Charles Suslowicz, Archanaa S Krishnan, Daniel Dinu, and Patrick Schaumont. Secure application continuity in intermittent systems. In *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8. IEEE, 2018.

[232] Igor Sysoev. nginx.

[233] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[234] Gang Tan et al. *Principles and Implementation Techniques of Software-based Fault Isolation*. Now Publishers, 2017.

[235] Xi Tan, Zheyuan Ma, Sandro Pinto, Le Guan, Ning Zhang, Jun Xu, Zhiqiang Lin, Hongxin Hu, and Ziming Zhao. Sok: Where's the "up"?! a comprehensive (bottom-up) study on the security of arm cortex-m systems. In *18th USeNIX WOOT conference on offensive technologies (WOOT 24)*, pages 149–169, 2024.

[236] Jack Tang and Trend Micro Threat Solution Team. Exploring control flow guard in windows 10. *Available at ht tp://blog. trendmicro. c om/trendlabssecurity-intelligence/exploring-control-flow-guard-in-windows*, 10, 2015.

[237] Texas Instrument. MSP430 Competitive Benchmarking. Application Report SLAA205B, 2006.

[238] Texas Instruments. Msp430fr596x,msp430fr594xmixed-signalmicrocontrollers. `https://www.ti.com/product/MSP430FR5969`, 2018. Last accessed: Feb. 23, 2025.

[239] Texas Instruments. Msp430fr58xx, msp430fr59xx, msp430fr68xx, and msp430fr69xx family user's guide. `https://www.ti.com/lit/ug/slau367p/slau367p.pdf`, 2020. Last accessed: Feb. 23, 2025.

[240] Texas Instruments, Inc. Fram faqs. `http://www.ti.com/lit/ml/slat151/slat151.pdf`, 2014. Last accessed: Feb. 23, 2025.

[241] The Clang Team. Control Flow Integrity. `https://clang.llvm.org/docs/ControlFlowIntegrity.html`, 2022. Last accessed: Feb. 23, 2025.

[242] The Clang Team. Safe Stack. `https://clang.llvm.org/docs/SafeStack.html`, 2022. Last accessed: Feb. 23, 2025.

[243] The Clang Team. Shadow Call Stack. `https://clang.llvm.org/docs/ShadowCallStack.html`, 2022. Last accessed: Feb. 23, 2025.

[244] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX security symposium (USENIX security 14)*, pages 941–955, 2014.

[245] Richard Townsend. Enhancing chromium's memory safety with armv9.

[246] Digital Trends. Hackers broke into a casino's high-roller database through a thermometer in the lobby fish tank. `https://www.digitaltrends.com/home/casino-iot-hackers-fish-tank/`, 2018. Last accessed: Feb. 23, 2025.

[247] Hoang Truong, Shuo Zhang, Ufuk Muncuk, Phuc Nguyen, Nam Bui, Anh Nguyen, Qin Lv, Kaushik Chowdhury, Thang Dinh, and Tam Vu. Capband: Battery-free successive capacitance sensing wristband for hand gesture recognition. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 54–67, 2018.

[248] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2. `http://www.trustedcomputinggroup.org/tpm-main-specification/`, 2011. Last accessed: Feb. 23, 2025.

[249] TrustedFirmware.org. About op-tee.

[250] European Union. General data protection regulation gdpr.

[251] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. Multi-tag: A hardware-software co-design for memory safety based on multi-granular memory tagging. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 177–189, 2023.

[252] Timothy Vaccarelli. Sha256-512, 2017.

[253] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[254] Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 17–32, 2016.

[255] Jaikumar Vijayan. Target attack shows danger of remotely accessible HVAC systems. `https://www.computerworld.com/article/2487452/target-attack-shows-danger-of-remotely-accessible-hvac-systems.html`, 2014. Last accessed: Feb. 23, 2025.

[256] Kuznetsov Volodymyr, Szekeres Laszlo, Payer Mathias, Candea George, and R Sekar. Code-pointer integrity. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[257] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[258] Q-Success W3Techs. Usage statistics of nginx.

[259] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, dec 1993.

[260] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan C Ward. Control-flow integrity for real-time embedded systems. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[261] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 352–369. IEEE, 2022.

[262] Wang Wei. Casino Gets Hacked Through Its Internet-Connected Fish Tank Thermometer. `https://thehackernews.com/2018/04/iot-hacking-thermometer.html`, 2018. Last accessed: Feb. 23, 2025.

[263] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *Proceedings 2019-Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2019.

[264] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. Sponge-based control-flow protection for iot devices. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 214–226. IEEE, 2018.

[265] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. Ink: Reactive kernel for tiny battery-less sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 41–53, 2018.

[266] Zheng Yu, Ganxiang Yang, and Xinyu Xing. Shadowbound: Efficient heap memory protection through advanced metadata management and customized compiler optimization. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7177–7193, 2024.

[267] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391. IEEE, 2017.

[268] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[269] Mingwei 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*, pages 91–100, 2015.

[270] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J Walls. Silhouette: Efficient protected shadow stacks for embedded systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1219–1236, 2020.

[271] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Good motive but bad design: Why ARM MPU has become an outcast in embedded systems. *arXiv preprint arXiv:1908.03638*, 2019.

[272] Xia Zhou, Jiaqi Li, Wenlong Zhang, Yajin Zhou, Wenbo Shen, and Kui Ren. Opec: operation-based security isolation for bare-metal embedded systems. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 317–333, 2022.

[273] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

Appendix A

PISTIS

A.1 List of instrumented instructions

The list of instrumented instructions is illustrated in Table A.1.

A.2 Applications descriptions

Our application test suite is composed of 13 different applications described in Table A.2. We categorised the apps based on the type of operations performed. Specifically, we distinguish between I/O-oriented applications (i.e. using peripherals and performing memory operations), computational applications (i.e. performing CPU-intensive operations), and a mix of the two. Our application selection covers multiple domains, comprising two cryptographic primitives, several mathematics functions, a machine learning algorithm, a synthetic benchmark and applications in other domains. On top of 3 custom-made applications, appositely designed to stress different subsystems of our reference MCU, we selected 4 open-source applications from public repositories and 6 benchmark applications from the TI MSP430 Competitive Benchmark [237]. The latter is a representative selection[1] of the applications proposed by TI, comprising both simple and complex mathematical operations, and a synthetic benchmark.

[1]The original TI Competitive Benchmark contains several redundant applications. We chose the most relevant subset containing approximately one application per type.

A.3 Formal proofs of theorem 1 and 2

A.3.1 Proofs for theorem 1

In order to prove Theorem 1, we formalised the three operations in nuXmv [47] (a state of the art symbolic model checker), and for each of the three formalisations we considered some Linear Temporal Logic (LTL) [165] properties aiming at proving the correctness of the operations. In this approach we codify the three different operations as a sequential program encoded in nuXmv in form of Single Static Assignment [57] (as it is typically done in compilers and in software model checking), specifying for each value of the program counter: `s0` before the implicit condition checking the respective access policy i.e. $AP_R(i, M)$, $AP_W(i, M)$ or $AP_X(i, M)$); `s1` if the access point condition holds; `s2` right after the real operation on the memory for reading, writing, or modifying the program counter if correct; `end` representing the location to jump to if the access policy is violated. The variable `i` is a free variable that models the address we aim at reading from, writing to, and jumping to respectively. Then we have the three memories `PM` (Primary Memory), `VM` (Volatile Memory), and `MMIO`. `v` is the value to write in the memory in the case of the $\texttt{Write}_{\texttt{sf}}$.

Listing A.2 contains the nuXmv code for the $\texttt{Read}_{\texttt{sf}}$. Here we model the $\texttt{Read}_{\texttt{sf}}$ with the `DEFINE ReadSV` that is a word of size `N+1` where the `N+1` bit is set to `0` if the access policy $AP_R(i, M)$ holds, and to `1` otherwise. The model is then complemented with three LTL properties. The first states that if the $AP_R(i, M)$ is always true, then the `state` can never take value `end`. The second property states that if the $AP_R(i, N)$ is always true, then the `N+1` is always `0` if the `state` is different from `s0` (i.e. the state before a $\texttt{Read}_{\texttt{sf}}$). Finally, the last LTL property states that if it is possible to reach a state where the violation of the $AP_R(i, N)$ holds, then it is possible to reach the state `end` (the state where the read has violated the access policy). In this model, the `i` variable can range over any possible value (there is thus an implicit universal quantification ($\forall$)).

Listing A.1 reports the output of running nuXmv (taken from `https://nuxmv.fbk.eu/`) on this file. The execution has been done on a Linux laptop. nuXmv was able to prove (or disprove) the three LTL properties in few seconds.

These results show that the first two properties hold, while the last one is violated (as expected) to indicate that the only possible way to reach the `end` state is to violate the $AP_R(i, M)$ in a $\texttt{Read}_{\texttt{sf}}$. In this case, nuXmv can generate a trace showing how to reach that state (in the run for the sake of presentation we disabled the extraction of

```
computer_shell > nuXmv -int -dcx pistis_read.smv
*** This is nuXmv 2.0.0 (compiled on Oct 14 2019)
....
nuXmv > go_msat
nuXmv > check_ltlspec_ic3 -i
....
-- LTL specification ( G APr -> G state != end)  is true
-- LTL specification ( G APr -> G (state != s0 -> ReadSV[32 : 32] = 0ud1_0))  is true
-- LTL specification ( F !APr -> F state = end)  is false
-- (trace generation was suppressed)
nuXmv > quit
```

Listing A.1: Run of NUXMV on the `pistis_read.smv` file to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$ instruction.

the counterexample, option `-dcx` at command line).

Similar considerations hold for the other two instructions. Listing A.3 is the NUXMV code for proving correctness of the $\texttt{Goto}_{\texttt{sf}}$, while the one for the $\texttt{Write}_{\texttt{sf}}$ instruction can be found available on the public repository of PISTIS [105].

A.3.2 Proof of theorem 2

Base case: There are four base cases each constituted respectively by:

I the `NOP` no-op instruction that does not perform any access to the memory neither for writing nor reading nor executing;

II the $\texttt{Read}_{\texttt{sf}}$;

III the $\texttt{Write}_{\texttt{sf}}$;

IV the $\texttt{Goto}_{\texttt{sf}}$.

The last three instructions preserve memory isolation as proved in Theorem 1. The `NOP` preserves memory isolation since it does not access memory neither for writing nor for reading nor for executing. Thus the single instruction program preserves memory isolation.

Step case: Let us assume a program P preserves memory isolation. Let $P' = P; \texttt{inst}$ be a program obtained by adding an instruction `inst` immediately after the last instruction of P. The possible instructions `inst` are: `NOP`, $\texttt{Read}_{\texttt{sf}}$, $\texttt{Write}_{\texttt{sf}}$, and $\texttt{Goto}_{\texttt{sf}}$. These instructions preserve memory isolation (given the base case and Theorem 1), thus given that P preserves memory isolation and the fact that the possible extension of the program P (i.e. the program P') also preserves the memory isolation, we can conclude that the theorem holds.

Table A.1: List of main MSP430 instructions with a brief description and whether they had been instrumented by PISTIS.

Instruction	Description	Instrumented	Why
ADC, ADCX	Add carry bit to destination	No ✗	Does not affect PISTIS
ADD, ADDX, ADDC, ADDCX	Add word to destination (w, w/o carry bit)	No ✗	Does not affect PISTIS
AND, ANDX	AND between source and destination	Yes ✓	Can unlock Flash controller
BIC, BICX, BIS, BISX, BIT, BITX	Clear, set, test bits of destination	No ✗	Does not affect PISTIS
BR	Jump to destination word	No ✗	Does not affect PISTIS
CALL	Call a function	Yes ✓	Can break application boundaries
CLR, CLRX, CLRC, CLRN, CLRZ	Clear destination or carry, negative, zero bits	No ✗	Does not affect PISTIS
CMP, CMPX	Compare source and destination	No ✗	Does not affect PISTIS
DADC, DADCX, DADD, DADDX	Add word to destination (w, w/o carry bit)	No ✗	Does not affect PISTIS
DEC, DECX, DECD, DECDX	Decrement, double decrement destination	No ✗	Does not affect PISTIS
DINT, EINT	Disable, enable interrupts	No ✗	Does not affect PISTIS
INC, INCX, INCD, INCDX	Increment, double increment destination	No ✗	Does not affect PISTIS
INV, INVX	Invert destination	No ✗	Does not affect PISTIS
JC, JHS, JEQ, JZ, JQE, JL, JMP, JN, JNC, JLO, JNZ, JNE	Jump to destination with a condition	Yes ✓	Can break application boundaries
MOV, MOVX	Copy source to destination	Yes ✓	Can break application boundaries, Can unlock Flash controller
NOP	No operation	No ✗	Does not affect PISTIS
POP, POPX, POPM	Pop value, values, from stack to register, registers	Yes ✓	Can break application boundaries, Can unlock Flash controller
PUSH, PUSHX, PUSHM	Push register, registers, to the stack	Yes ✓	Can unlock Flash controller
RET, RETI	Fetch return address from the stack	Yes ✓	Can break application boundaries
RLA, RLAM, RLAX, RLC, RLCX, RRA, RRAM, RRAX, RRC, RRCX, RRUM, RRUX	Rotate destination	No ✗	Does not affect PISTIS
SBC, SBCX	Subtract borrow bit from destination	No ✗	Does not affect PISTIS
SETC, SETN, SETZ	Set carry, negative, zero bits on status register	No ✗	Does not affect PISTIS
SUB, SUBX, SUBC, SUBCX	Subtract source to destination (w, w/o carry bit)	Yes ✓	Can unlock Flash controller
SWPB, SWPBX	Swap bytes of destination	No ✗	Does not affect PISTIS
SXT, SXTX	Extend sign of destination	No ✗	Does not affect PISTIS
TST, TSTX	Test destination	No ✗	Does not affect PISTIS
XOR, XORX	XOR between source and destination	Yes ✓	Can unlock Flash controller

Table A.2: An overview of the main functionalities of the applications used in evaluating PISTIS.

App	Category	Domain	Source	Description
SerialMSP	I/O	Inter-device communication	Custom made	Set up an UART connection with a terminal and initiate the download of chunks of data to be stored on the RAM.
CopyDMA	I/O	Data migration / backup	Custom made	Initiate the migration of chunks of data from FLASH to RAM employing both CPU and DMA. The transferred data is checked for errors to verify its integrity.
XorCypher	Computational	Crypto	Custom made	Encodes various data streams with a custom-XOR stream cipher, saving the new encoded data to protect its confidentiality.
Bitcount	Computational	Benchmark	OpenSource https://github.com/embecosm/mibench	Perform different bit-based intensive computations over data arrays to test the computational power of the CPU.
SHA-256	mix	Crypto	OpenSource https://github.com/amosnier/sha-2	Compute the hash of various strings using the SHA-256 algorithm, storing it alongside the strings for integrity protection.
ML-acc	mix	Machine Learning	OpenSource https://github.com/CMUAbstract/dino	Train an activity recognition machine learning model to classify new accelerometer measurements into "shaking" and "still" activity.
PrimeFactor	mix	Mathematics	OpenSource https://github.com/TheAlgorithms/C	Calculate the prime factorisation for various input numbers to test the computational power of the CPU.
TI-32bitMath	Computational	Mathematics	TI Benchmark	Perform several 32-bit math operations on various input data.
TI-16bitSwitch	Computational	Operational	TI Benchmark	Test various switch cases using different 16-bit data.
TI-8bitMatrix	I/O	Operational	TI Benchmark	Backup the data from 8-bit based matrixes in RAM.
TI-MatrixMul	Computational	Mathematics	TI Benchmark	Perform different multiplications between matrixes in RAM, saving the new data.
TI-firFilter	mix	Signal processing	TI Benchmark	Benchmark an FIR filter of order 17 with various input data contained in arrays of 16-bit values.
TI-dhrystone	mix	Benchmark	TI Benchmark	Dhrystone synthetic benchmark

```
-- To run it: "shell > nuXmv -int pistis_read.smv". At the nuXmv prompt issue following commands: "go_msat;
     check_ltlspec_ic3 -i; quit" --
MODULE main
  VAR PM   : array word[32] of word[32]; -- Persistent memory
  VAR VM   : array word[32] of word[32]; -- Volatile memory
  VAR MMIO : array word[32] of word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM}; -- kind of memory
  VAR state : {s0, s1, s2, end}; -- Possible values of the PC
  VAR PC : word[32]; -- The program counter;
  VAR v : word[32]; -- Value to write
  VAR i : word[32]; -- Address to read from/write to

  -- Memory layout as mandated by the policy
  VAR EPadd : array word[3] of word[32]; -- List of entry points
  DEFINE utc_b  := 0h32_00000010;
  DEFINE utc_e  := 0h32_00000100;
  DEFINE aim_b  := 0h32_00001000;
  DEFINE aim_e  := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b  := 0h32_00001000;
  DEFINE adm_e  := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;

  INVAR
    0h32_00000000 < utc_b & utc_b < utc_e & utc_e < aim_b &
    aim_b < aim_e & aim_e < arom_b & arom_b < arom_e &
    arom_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < utdm_b & utdm_b < utdm_e &
    utdm_e < adm_b & adm_b < adm_e & adm_e < 0h32_FFFFFFFF;
  INVAR
    0h32_00000000 < mmio_b & mmio_b < mmio_e & mmio_e < 0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
   (((mem_k = _PM  ) -> ((arom_b <= i) & (i <= arom_e))) &
    ((mem_k = _VM  ) -> ((adm_b  <= i) & (i <= adm_e ))) &
    ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APw :=
   (((mem_k = _VM  ) -> ((adm_b  <= i) & (i <= adm_e))) &
   ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APx :=
   ((mem_k = _PM  ) & (EP | ((aim_b  <= i) & (i <= aim_e))));

  DEFINE EP := (((utc_b <= i) & (i <= utc_e)) &
    ((i = READ(EPadd, 0d3_0)) | (i = READ(EPadd, 0d3_1)) |
     (i = READ(EPadd, 0d3_2)) | (i = READ(EPadd, 0d3_3)) |
     (i = READ(EPadd, 0d3_4)) | (i = READ(EPadd, 0d3_5)) |
     (i = READ(EPadd, 0d3_6)) | (i = READ(EPadd, 0d3_7))));

  -- Read_sf(M, i) := if (APr(i,M)) return M[i] else goto end;

  ASSIGN
    init(state) := s0;
    next(state) := case
                     state = s0 & APr : s1;
                     state = s1 & APr : s2;
                     state = s2       : s2;
                    TRUE              : end;
                  esac;

   DEFINE ReadSV := case
     state = s0 & APr : 0d33_0;
     state = s1 & APr : case
                  mem_k = _PM   : 0d1_0 :: READ(PM, i);
                  mem_k = _VM   : 0d1_0 :: READ(VM, i);
                  mem_k = _MMIO : 0d1_0 :: READ(MMIO, i);
              TRUE              : 0h33_FFFFFFFF;
                esac;
     state = s2  : 0d1_0 :: 0h32_FFFFFFFF;
     state = end : 0d33_0;
     TRUE        : 0d33_0;
   esac;

  -- If the APr is always true, then there is not a possibility to reach the end state.
   LTLSPEC
    G(APr) -> G(state != end)

  -- If the APr is always true, and we are in any state different from s0, then the error flag bit is always
       0
  LTLSPEC
     G(APr) -> G(state != s0 -> ReadSV[32:32] = 0d1_0)

  -- If APr is violated, then the state eventually become end, i.e. end is only reachable if APr is violated
  LTLSPEC
     F(!APr) -> F (state = end)
```

Listing A.2: The encoding to prove the correctness of the $\texttt{Read}_{\texttt{sf}}$

```
-- To run it: "shell > nuXmv -int pistis_goto.smv". At the nuXmv prompt issue following commands: "go_msat;
      check_ltlspec_ic3 -i; quit" --
MODULE main
  VAR PM   : array word[32] of word[32]; -- Persistent memory
  VAR VM   : array word[32] of word[32]; -- Volatile memory
  VAR MMIO : array word[32] of word[32]; -- MMIO
  VAR mem_k : {_PM, _MMIO, _VM}; -- kind of memory
  VAR state : {s0, s1, s2, end}; -- Possible values of the PC
  VAR PC : word[32]; -- The program counter;

  VAR v : word[32]; -- Value to write

  VAR i : word[32]; -- Address to read from/write to

  -- Memory layout as mandated by the policy
  VAR EPadd : array word[3] of word[32]; -- List of entry points
  DEFINE utc_b  := 0h32_00000010;
  DEFINE utc_e  := 0h32_00000100;
  DEFINE aim_b  := 0h32_00001000;
  DEFINE aim_e  := 0h32_00010000;
  DEFINE arom_b := 0h32_00100000;
  DEFINE arom_e := 0h32_01000000;
  DEFINE utdm_b := 0h32_00000010;
  DEFINE utdm_e := 0h32_00000100;
  DEFINE adm_b  := 0h32_00001000;
  DEFINE adm_e  := 0h32_00010000;
  DEFINE mmio_b := 0h32_00000010;
  DEFINE mmio_e := 0h32_00000100;
  DEFINE END    := 0h32_10000000;

  INVAR
   0h32_00000000 < utc_b & utc_b < utc_e & utc_e < aim_b &
   aim_b < aim_e & aim_e < arom_b & arom_b < arom_e &
   arom_e < 0h32_FFFFFFFF;
  INVAR
   0h32_00000000 < utdm_b & utdm_b < utdm_e &
   utdm_e < adm_b & adm_b < adm_e & adm_e < 0h32_FFFFFFFF;
  INVAR
   0h32_00000000 < mmio_b & mmio_b < mmio_e & mmio_e < 0h32_FFFFFFFF;

  -- Access policy
  DEFINE APr :=
    (((mem_k = _PM  ) -> ((arom_b <= i) & (i <= arom_e))) &
     ((mem_k = _VM  ) -> ((adm_b  <= i) & (i <= adm_e ))) &
     ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APw :=
    (((mem_k = _VM  ) -> ((adm_b  <= i) & (i <= adm_e))) &
     ((mem_k = _MMIO) -> ((mmio_b <= i) & (i <= mmio_e))));

  DEFINE APx :=
    ((mem_k = _PM  ) & (EP | ((aim_b  <= i) & (i <= aim_e))));

  DEFINE EP := (((utc_b <= i) & (i <= utc_e)) &
    ((i = READ(EPadd, 0d3_0)) | (i = READ(EPadd, 0d3_1)) |
     (i = READ(EPadd, 0d3_2)) | (i = READ(EPadd, 0d3_3)) |
     (i = READ(EPadd, 0d3_4)) | (i = READ(EPadd, 0d3_5)) |
     (i = READ(EPadd, 0d3_6)) | (i = READ(EPadd, 0d3_7))));

  -- goto_sf(M, i, v) := if (APx(i,M)) PC = i else goto end;

  ASSIGN
    init(state) := s0;
    next(state) := case
                     state = s0 & APx : s1;
                     state = s1 & APx : s2;
                     state = s2       : s2;
                     TRUE             : end;
                   esac;
 INIT
  PC != END;
 ASSIGN
  next(PC) := case
                state = s0 & APx : PC;
                state = s1 & APx : i;
        state = s2       : PC;
        TRUE             : END;
              esac;

  -- If the APx is always true, then there is not a possibility to reach the end state.
  LTLSPEC
    G(APx) -> G(state != end)

  -- If the APx is always true, then the PC never assumes value END
  LTLSPEC
     G(APx) -> G (state != s0 -> PC != END)

  -- If APx is violated, then the state eventually become end, i.e. end is only reachable if APx is violated
  LTLSPEC
     F(!APx) -> F (state = end)
```


Listing A.3: The encoding to prove the correctness of the $\texttt{Goto}_{\texttt{sf}}$

Appendix B

TAGShield

B.1 LLVM

The LLVM project is an open-source compiler toolchain extensively used in both industry and academia for its versatility and extensibility. At its core, LLVM is mainly organised in passes, i.e. single functional components that carry out a specific task in the compilation process, transforming, analysing and/or optimising the code. Passes can be modified, extended or integrated with new ones, leading to a modular and extensible infrastructure.

To bolster the cross-compatibility and interoperability of passes, LLVM offers an expressive Intermediate Representation (IR) language [202] on top of which passes can operate. This language is based on a low-level comprehensive semantic that allows the source code in most languages to be converted into IR. To facilitate the reading of the following sections, we briefly describe four IR instructions: alloca, load, store and call.

The *alloca* instruction (`%0 = alloca <type>, align <alignment>`) allocates new memory on the stack for the given type, returning a pointer to it. The *load* (`%0 = load <type>, ptr <ptr>, align <alignment>`) and *store* (`store <ptrSrc>, ptr <ptrDst>, align <alignment>`) instructions deal with memory, respectively loading a value from memory into a variable, and storing a variable into the memory pointed by a pointer. Finally, the *call* (`call <fun>(<args,...>)` instruction invokes a specific function with some arguments. Listing B.1 shows an example of the four instructions.

```
//Save two int on the stack
%a = alloca i32, align 4
%b = alloca i32, align 4

//Load the value in %b into a new var %0
%0 = load i32, ptr %b, align 4

//Store %0 into the memory pointed by %a
store %0, ptr %a, align 4

call @multiply(%0,%a)
```

Listing B.1: Example of '`int a = b;`' in LLVM IR.