Prototyping Memory Integrity Tree Algorithms for Internet of Things Devices

Ionuț Mihalcea

Technical Report

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

Project Submission Declaration

UNIVERSITY OF LONDON - MSc INFORMATION SECURITY

Dissertation project title	Prototyping Memory Integrity Tree Algorithms for Internet of Things Devices
Name	Ionuț Alexandru Mihalcea
Student number	160140639

ANTI-PLAGIARISM DECLARATION

I declare that this dissertation is all my own work, and that I have acknowledged all Quotations from the published or unpublished works of other people.

I declare that I have also read the statement on plagiarism in the General Regulations for Awards at Graduate and Masters Levels for the MSc in Information Security and in accordance with it I submit this project report as my own work.

Signed: 

Dated: 31/03/2022

Summary

The technological progress of the last decade has led to a wealth of small computing platforms that can be easily integrated in everything from medical implants to industrial machinery. Not only are more and more objects being augmented with sensing, computation, and actuation capabilities, but many of these new devices are being engineered and deployed to serve as internet endpoints, essentially creating an Internet of Things (IoT).

This vast new ecosystem grew out without an emphasis on security, leading to myriads of attacks – from ransomware to botnets of millions of devices. While the industry has been making progress towards plugging the vulnerabilities underlying these attacks, the issue forming the core of this thesis – security in the face of physical attacks – has seen far less attention in the IoT community. The case is made here for defending against such threats by analysing IoT platform architectures, deployment platforms, and various best-practice guidelines. Despite not being as easy to perform as other types of attacks, physical attacks typically have no barrier and can completely compromise high-profile targets.

Multiple mitigation approaches exist against physical attacks, however they come with high cost, complexity, and performance degradation burdens. Among them, the focus of this work falls on cryptographic memory protection, and on bringing down the performance barriers which prevent wider adoption of this technology. Thus, the evolution of such cryptographic mechanisms is reviewed with emphasis on performance optimization techniques.

The first contribution presented here represents new data structures and a new architectural approach to enforcing cryptographic memory protection. These new mechanisms extend existing techniques, and recognize and anticipate progress in device manufacturing. In particular, the possibility of benefiting from a memory block more tightly coupled with the processor is explored.

The second contribution consists of a software implementation of our proposed techniques. Built using an existing memory protection prototype (developed during previous research projects within Arm Ltd.) as a baseline, the new techniques have been implemented, tested, and benchmarked to understand their effect on the performance of this enhanced system, relative to an unprotected version of it. Other promising techniques proposed previously in literature, but which were not part of the baseline prototype, were also implemented and tested in a similar way.

The results of this benchmarking process are given, showing major improvements from some of the proposed techniques, at a relatively low cost. Combinations and trade-offs between these optimizations are also discussed.

Acknowledgements

I would like to thank Prof. Konstantinos Markantonakis for his guidance during my work on this thesis.

I would also like to thank those forming my support pillars within Arm Ltd: Dr. Roberto Avanzi for his mentoring, technical expertise, and constant pursuit of perfection; Paul Howard for his kindness and unwavering moral support; and Dr. Andreas Sandberg for his help with understanding and working with gem5.

This thesis would not have been completed without the constant support and encouragement from my parents, Liliana and Săndel, and from my partner Цветина.

Table of Contents

1. Introduction

1.1. Motivation

The Internet of Things (IoT) is a technological trend that aims to close the gap between the real and virtual worlds. This is done by embedding computing platforms in physical objects such as household items like kettles or printers, or by dispersing them in the environment, such as on farm fields. The computing platforms can then be connected to the Internet and with each other, allowing remote sensor readings and actuation and thus enabling complex use cases that scale to internet proportions. A few of the fields where IoT is already building momentum are home automation, industrial monitoring, medical data collection and analysis, and many others [1]. The value boost brought by such technologies is well established, but this value comes at increased risks – general purpose computing and Internet connectivity in cyber-physical systems opens the door for cyberattacks that are all too common for traditional computers. A Denial of Service (DoS) attack is more consequential when it targets internet connected pacemakers than when targeting a website selling pacemakers. The world is, therefore, depending more and more on the correct operation of IoT endpoints.

By the nature of their functionality, IoT devices are located where there is a need for sensing or actuation, with many of these locations having little or no physical security. This characteristic creates a new dimension to the task of protecting these devices, as attackers can be assumed to have unmitigated physical access to them. This enables traditional attack vectors that abuse networking components such as Heartbleed [2] which exploited an issue in a widespread Transport Layer Security (TLS) protocol implementation. However, a new attack avenue that is typically difficult to exploit in a corporate office or in a datacentre – physical attacks – is also made trivial by the deployment patterns of IoT nodes. Physical attacks leverage access to the device to extract data from it or to affect its operation. This project will focus on one such type of attack – memory probing – where the attacker extracts or modifies data stored in primary memory. More specifically, the project will explore one protection mechanism against such attacks – memory encryption.

Cryptography provides many of the mechanisms used to protect data across networks, between internet endpoints. Safely browsing, using credentials or online shopping would be impossible without encryption hiding the contents of the packets or without digital signatures to help certify the identity of remote servers. Similarly, data persisted on hard disks or other non-volatile storage is frequently encrypted, including in smartphones [3]. Volatile memory, however, is generally left in the clear, based on the assumption that extracting the information straight from the chip or snooping on the bus connecting it to the processor are too difficult. This is even though volatile memory usually contains the cryptographic keys used for network protocols, which could give access to the victim's entire network. In the case of IoT devices, the decreasing cost and risk of an attack could easily be outweighed by taking over a cyber-physical system.

Despite an apparent need to secure primary memory, simply deploying off-the-shelf cryptographic solutions can lead to a decrease in performance that makes the technology completely unappealing. A balance must therefore be struck between usability and security, which this project aims to tackle with a distinct focus on IoT devices.

1.2. Scope and objectives

My goal in this project is to identify, prototype and evaluate novel mechanisms for protecting IoT devices from physical attacks targeting their memory. The category of mechanisms I am interested in is that of cryptographic engines offering confidentiality and integrity for physically exposed data. Such engines are implemented along with a processor in an integrated circuit (IC) and protect any data leaving the IC package. An example of this technology is represented within Intel's Software Guard Extensions (SGX), though SGX goes beyond defending against physical attacks and into memory access control. The broader field of memory protection (including various access control mechanisms) will be reviewed for completeness in later chapters.

Even more specifically, my project will look at the data structures and algorithms that mitigate the risk of a replay attack, where a chunk of memory is replaced with an older version of data stored in the same place. The consequences of such attacks, as well as the techniques used to thwart them will be thoroughly investigated in chapters **Error! Reference source not found.** and 4. Despite being a seemingly niche topic, replay attack mitigations are the most heavyweight mechanisms within the cryptographic memory protection toolbox and would therefore have an oversized impact on IoT devices. Thus, it is important to establish not only which implementations are most suitable, but also in what device classes and at what production and operational cost.

1.2.1. Project collaboration

This project is done with support from Arm Ltd. in the form of hardware used to develop and test the prototypes, as well as a software base on which I have built the techniques described in this report. This software base represents previous work done within the Arm research group, in particular the work done by David Schall for his Master thesis at the University of Kaiserslautern [4]. David's work focused on similar optimization techniques that aimed at improving the performance of integrity and replay protection (IARP) mechanisms. A more thorough description of his results and of the model that I inherited from him is given in chapter 6.

1.2.2. Objectives

- To establish the categories of IoT devices that might benefit from memory protection against physical attacks.
- To set forth IoT-specific characteristics based on which security mechanisms can be evaluated.
- To present the need for and scope of cryptographic memory protection.
- To identify and analyse novel integrity and replay protection algorithms and techniques, based on the relevant characteristics.
- To prototype and evaluate these algorithms to determine their feasibility in the case of different classes of IoT devices.

2. The Internet of Things

The evolution of computing since mid-20th century has been a continuous driver of societal change. From leisure activities such as video games, the bleeding edge science of the Large Hadron Collider to the sequencing of the human genome, faster and cheaper computing platforms have led to developments in a myriad of areas. It was networking, however – the physical and logical coupling of an increasing number of nodes – that has made the deepest mark on how people interact with the world and with each other. The slow, static content of the World Wide Web (WWW) era and its Internet Relay Chats (IRC) made way to the highly interactive Web 2.0, home of social media and rich content streaming. Cloud computing has been a core driver for this expansion, bringing cheap hosting and easy deployment of public-facing websites. The massive computing capacity created for storing and service content opened the door for analysis of data at scales impossible before. Web 3.0 was built on this infrastructure – a new dimension to the Internet, where the large amounts of information generated by users can be processed and fed back into the system, for a richer browsing experience. The Internet has thus evolved towards an environment where data handling is centralised through cloud providers relying on economies of scale. This topology is due to become more decentralised, with the edges forming a tighter integration between the virtual and real worlds. The Internet of Things comes in as an orthogonal component to the traditional human-centric networks by pulling our environment in the virtual world. Arrays of sensors and actuators suddenly become endpoints able to interpret and control their surroundings remotely and automatically. Looking further into the future, deployment of 5G and 6G telecommunication networks that enable ethernet-level speeds for wireless connections is sure to facilitate denser arrays of devices.

As this integration of technology into our day to day lives keeps deepening, the threat posed by security issues within the computing realm also grows in complexity and nuance. The purpose of this thesis is to investigate technical solutions to a specific type of vulnerabilities that is more easily exploitable in IoT devices. The following sections of this chapter describe the architecture of typical IoT systems, and the types of devices involved.

2.1. Architecture of an IoT system

Even though IoT has become a trend on its own, the endgame of IoT is not the network of “Things”, but rather the services built on top of them – be it data gathering and analysis, industrial control, smart infrastructure, or many other systems already in use. A generic system diagram showing the

layering of device types can be seen in Figure 1 below. The following sections will describe each layer in more depth.

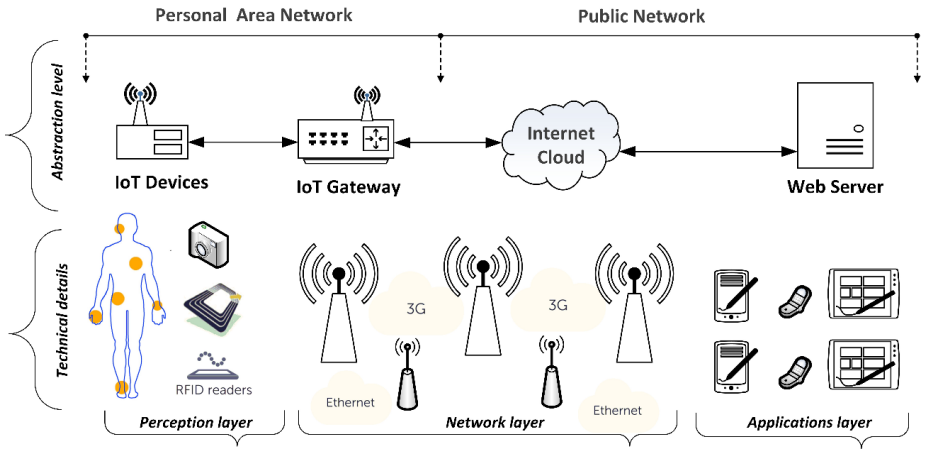

Figure 1 - High level architecture of an IoT system [127]

2.1.1. Perception layer

The perception layer is composed of endpoints that bridge between the virtual and real worlds. Devices in this layer represent a large diversity of computing, communication, sensing, and actuation capabilities as they tackle vastly different tasks in different environments. Most devices at this level can be considered constrained, which [5] defines as: "[nodes] where some of the characteristics that are otherwise pretty much taken for granted for Internet nodes at the time of writing are not attainable, often due to cost constraints and/or physical constraints on characteristics such as size, weight, and available power and energy". These limitations then cascade into constraints on the computing system, such as processor frequency, amount of available memory, or speed of communication links. The constraints relevant to IoT endpoints and the trade-offs that result from them are deeply tied to the problem they are trying to solve and the environment they operate in. This variety has led the available platforms to stretch from tiny boards such as EMBIT EMB-Z2530PA that measures 29 x 22 mm and uses an 8-bit single-core processor, to Single Board Computers (SBC) such as the Raspberry Pi [6] that runs a 64-bit quad core processor on a 40 x 55 mm board.

2.1.2. Network layer

The network layer is a bridge between the perception and application layers, starting with gateways that support both the protocols of WSN nodes and those used in traditional internet endpoints such as IP, TLS, or Transmission Control Protocol (TCP), followed by the traditional internet infrastructure of routers and switches.

In some deployments, the gateways servicing endpoints also act as sensors, or as lightweight processing platforms – a trend known as edge computing [7]. Under this paradigm nodes that are not capable of performing their own data processing can offload it to a gateway. Such processing at the edge also lifts some of the burden of moving the increasing amount of sensor data from network edges to datacentres. The need to place gateways within low-power communication range of sensing nodes means that the gateways are also more prone to being physically exposed. Thus, such devices are an attractive target for attacks that exploit physical access.

2.1.3. Application layer

The application layer consists mostly of powerful servers in the cloud that aggregate, process, and organise data gathered from endpoints. Energy efficiency and device cost is usually traded off at this level for performance and throughput, putting them in deep contrast with devices in the perception layer.

2.2. Device characteristics

This project focuses on the perception layer and the first level gateways that link it to the internet, as these devices face the biggest risk of physical attacks based on their typical deployment circumstances. This section takes a closer look at their hardware characteristics and at the constraints faced by engineers designing, building, and deploying them. The two device features of particular interest for this project are the processor and the memory system, and the interaction model between them.

2.2.1. Processor

As hinted at in previous sections, high-end IoT platforms have a need to balance the same set of competing requirements as all other computing platforms, but with a strong tilt towards energy efficiency and cost. Producing implementations that offer high performance on such tight power budgets has been of growing interest in the semiconductor industry.

Processors – commonly called Central Processing Units (CPU) – are the core component of any computing platform. They take binary-encoded instructions from memory, decode them, and perform the operations they represent, such as arithmetic operations or manipulations of data located in memory. CPUs vary greatly within the IoT space in terms of their speed, complexity, physical size of implementation, power, and so on. Some of these characteristics are inadvertently linked – a complex CPU will most likely require a larger physical footprint in the IC.

A source of complexity when matching up processors with applications is that of physically packaging the processor implementation. Typical desktop PCs and laptops, for example, contain a motherboard – a printed circuit board (PCB) that holds and connects discrete integrated circuit (IC) components, including the processor and Random-Access Memory (RAM). For such designs, the processor has its own packaging, often including supporting components such as caches in what is called a microprocessor.

On the other hand, the need for small power efficient devices has created a market for ICs that contain not only the processor but also RAM, read only memory (ROM), and peripherals like universal asynchronous receiver-transmitter (UART) modules. The dedicated peripherals included within the IC are usually for interaction with sensors or actuators (e.g. pulse width modulation modules for controlling motors) or for communication (e.g. Ethernet modules) [8]. These are called microcontroller units (MCU) and fit in a package of similar size to a microprocessor (or even smaller), tuned to require only a fraction of the power usage. Such a small physical footprint, however, leaves little room for adjustments – the processor implementation cannot be afforded a large area or an increased cut of the power budget without trading off other features within the package (like, for example, the amount of ROM available). This reduced space and power budget puts a cap on what other components can be sensibly added to the system. Heavyweight security features that not only occupy a large proportion of the IC but also have a considerable impact on the execution speed can easily outweigh the benefits they could bring to an MCU [9].

Among the processor architectures present in the market, Arm holds a dominant position that was obtained through a development process prioritizing power efficiency [8]. Among the low-powered devices, particularly MCUs, the Arm Cortex-M processor family is prevalent. For example, among the low-end devices sampled in [9], the majority rely on Arm Cortex M devices, primarily Cortex M3 and Cortex M0. Similarly, on the high-end of the performance spectrum, Arm offers the Cortex-A cores that are frequently deployed in multi-core configurations. As an example, Raspberry Pi 4 [6], the latest release in the series, relies on a quad-core A72 processor.

2.2.2. Primary storage

Primary storage is a term encompassing the whole hierarchy of memory technologies used in computing platforms for fast access at system runtime. In comparison, secondary storage covers the persistent, long term storage technologies such as hard disks, which also suffer from long access times. Primary storage consists of: RAM, ROM, processor caches and flash memory. While all these play important roles in IoT devices, RAM and caches play central roles in cryptographic memory protection as will be detailed in chapter 4. This section therefore looks at RAM and processor caches in the context of IoT. Figure 2 below shows the available array of technologies, starting with largest yet slowest ones at the bottom, and finishing with the fastest yet most expensive at the top.

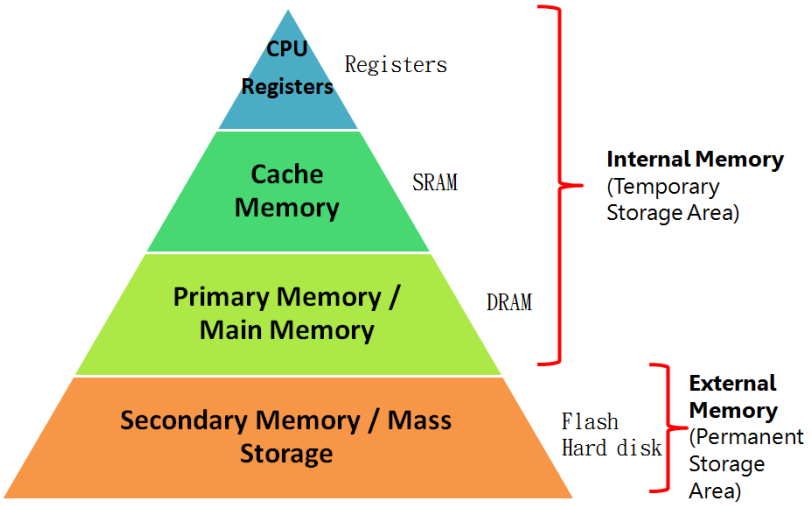

Figure 2 - Computer memory hierarchy [10]

This thesis is concerned with security services applied between the caches and main memory. DRAM memory is usually off-chip (from the perspective of the processor) and is usually orders of magnitude larger than the on-chip memory. Main memory is used to store the full state of the software executing on the device, as well as a medium of communication between processes and/or device components. In the lack of any specialised hardware, all sensitive data that is handled by the device ends up residing in primary memory.

Processor caches act as a buffer between CPU and memory with the main goal of cutting down the latency of memory-related operations. Typically, the caches themselves are arranged in a hierarchy with multiple levels, with the lower numbered layers closer to the CPU. Thus, in a processor with 2 cache levels, the level closest to the CPU would be denoted L1 and the subsequent level L2. The same trade-off of characteristics between RAM and caches can be observed between cache levels as well, with L1 being the smallest and fastest level, while the last level is the largest and slowest. Of greater interest with respect to cryptographic memory protection is the last level of cache (LLC), as its structure and operating details must be aligned with the module offering memory protection. The

LLC essentially sits at the virtual inner border of the chip - all data in LLC or any level closer to the CPU is trusted to be valid while all data beyond LLC (i.e. in DRAM or secondary storage) is considered potentially invalid and must be verified.

Any cache-specific properties relevant to the memory protection techniques discussed later in the thesis will also be covered at that point.

2.3. Perception device classes

As mentioned previously, this section deals with the characteristics of devices typically found in the perception layer. The disparity between high-end and low-end specifications is wide enough to warrant treating them as separate categories. Information presented in this section depends heavily on the work done in [9] and on IETF Request for Comments (RFC) 7228, "Terminology for Constrained-Node Networks" [5].

Low-end devices represent the most heavily constrained endpoints, stretching from Radio-frequency identification (RFID) tags to devices based on small MCUs. Nodes in this category are generally unable to run full-fledged operating systems and thus are restricted to running limited, highly specific sensing or actuation applications. In terms of technical specifications, 8- or 16-bit processors are the most common, with code storage space in the hundreds of kibibytes (KiB) and random-access memory (RAM) in the tens of KiB. Embedded flash storage and static RAM (SRAM) are commonly included in the same package as the processor. This turns out to be an indirect mitigation against attacks targeting the communication between RAM and processor as such attacks would be much more difficult than when the two components communicate over an inter-chip bus [11]. If an attacker can read signals within the chip enclosure, then they could bypass any data encryption performed within the chip as well.

Middle-end devices are less constrained than low-end devices, capable of running (stripped-down) operating systems and thus of performing more complex computational tasks. Despite these improvements, limitations on data storage and memory size (both in the hundreds of KiB) are still major bottleneck points for such nodes. Some of these devices offer larger amounts of RAM in the form of external double data rate dynamic RAM (DDR-DRAM).

High-end devices are the closest to resembling a traditional personal computer (PC) in terms of their capabilities – and thus are typically called Single Board Computers (SBC). A 32- or 64-bit multicore processor coupled with gibibytes (GiB) of external storage and RAM and potentially a graphical processing unit (GPU) allows them to run operating systems and heavy workloads such as machine learning algorithms. Due to their increased capabilities, such platforms can be used to offload computation from more constrained devices or as gateways to the rest of the internet. Their value to attackers is therefore increased by the richness of data being handled and by the greater importance afforded to them within the network topology. For mid- and high-end devices the power budgets hover in single digit Watts and cost below a hundred dollars.

Table 1 - Typical characteristics for the different IoT device classes

Device category	**Low end**	**Middle end**	**High end**
Processor	8- or 16-bit, single core	16- or 32-bit, single core	32- or 64-bit, single or multi core

Storage (e.g. flash)	Hundreds of KiB, embedded	High hundreds of KiB (embedded) to GiB (external)	Hundreds of MiB to GiB, external
RAM	Tens of KiB, embedded	Hundreds of KiB (embedded) to hundreds of MiB (external)	Hundreds of MiB to Gib, external

As mentioned earlier, physical attacks on chip memory can be considered orders of magnitude more difficult when the processor and memory share the same silicone substrate. Moreover, the cost, power, and performance constraints specific to low-end devices makes them poor candidates for bulky security features. Mid- and high-end devices can therefore be considered the most likely targets and thus the focus of this project. Subsequent sections will thus be limited to the devices that fall roughly into these categories.

2.3.1. Reference device characteristics

No specific device will be used as a reference platform for this project – instead, a device profile is given below which should cover the classes of devices that could benefit from dedicated memory security features.

The target device can run a modern operating system (OS), with a processor that typically has two levels of caches working as a bridge to main memory. Main memory consists of several GiB of DRAM accessible over an external bus. The processor must (at least) resemble, in its configuration and speed, processors typically found in mid- and high-end IoT devices.

2.4. Threat landscape

Assessing the dangers faced by IoT devices in the field requires an understanding of concepts central to security - assets, threats, vulnerabilities - and mapping them within the given context. Assets represent goods that have some value to the stakeholders of the system. A threat is any activity or occurrence that puts some asset(s) at risk. Vulnerabilities are flaws that could be exploited, allowing a threat to affect an asset. The following sections describe the assets relevant to this project, the vulnerabilities that could leave them exposed, and the mitigations that have been researched and implemented thus far. Figure 3 below shows a generic threat model for an IoT device. This thesis is mostly concerned with the threats posed by Physical Attackers – more specifically those targeting the communication between Kernel and Keys in Use, and/or the Memory block directly.

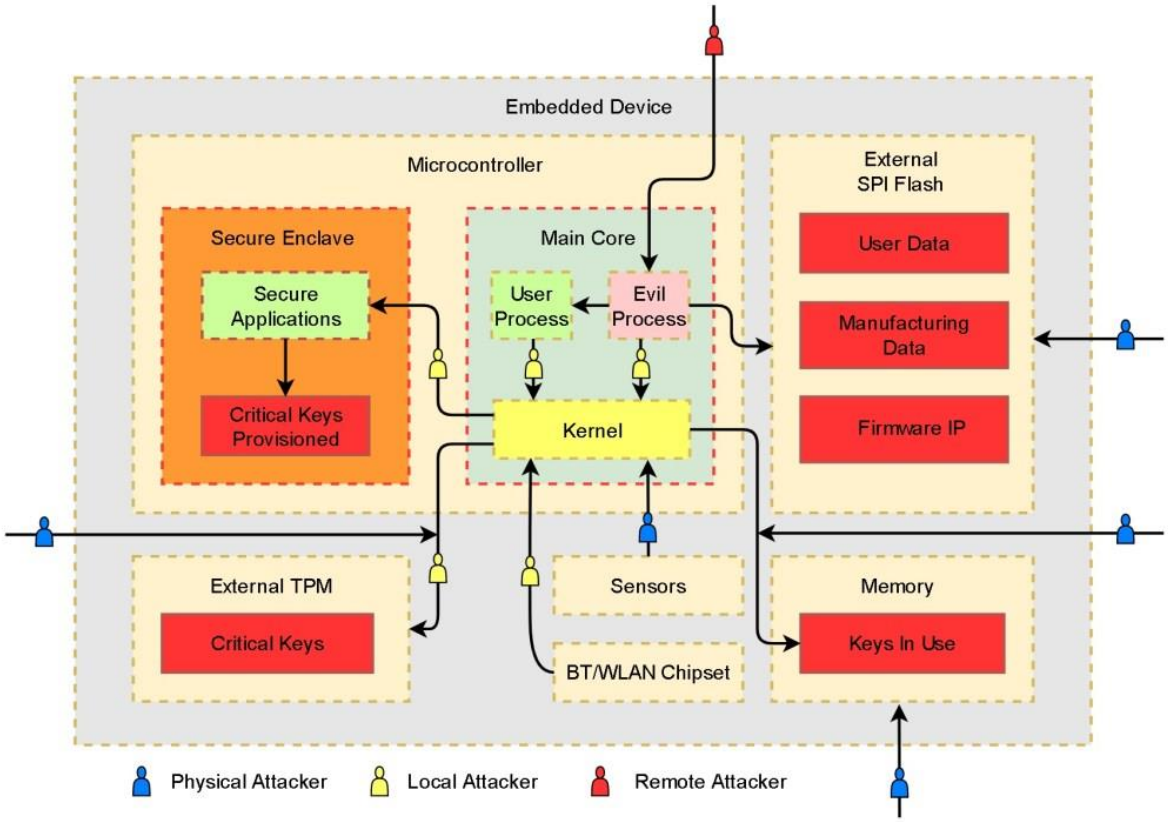

Figure 3 - Threat model for a generic IoT device [12]

2.4.1. Assets

Assets are a broad and elusive category to define and value, especially in physically complex systems such as IoT deployments: they include both abstract concepts such as customer trust in the overall service, virtual concepts such as network connectivity or sensor data, and physical goods like the devices themselves. Many of them, however, either have complicated relationships with or are irrelevant for the mechanisms discussed within the scope of this project. This discussion will thus be limited to the assets that can be directly defended through memory protection mechanisms found on the device. These can be divided roughly into:

- **Sensor data and actuator commands**: a natural asset to consider in a system based on a network of sensors and actuators. This information is both what gives the system its value, and what helps establish the outcome that the system is intended to inform or perform. For example, temperature readings in a geographical area can help authorities determine when to send out fleets to defrost roads. Therefore, the value of this asset is tied to the value of any other assets that depend on the correct operation of the system. Continuing the previous example, the cost of not defrosting streets can easily include car accidents and even loss of life.
- **Cryptographic keys and other credentials**: used within the IoT system have an indirect purpose of protecting the rest of information exchanged between parties. Be it raw or aggregate sensor data, or software updates, data flows between nodes in the system must be protected - generally through cryptographic means. Though all assets in this category are used to secure other assets, there are multiple ways in which the entities involved get access to specific keys or credentials. Some are generated or exchanged through networking

protocols that allow the two parties to establish a new key each time they communicate. Others are long-term (or even permanent), generally stored in special hardware modules that deter unnecessary or malicious exposure. These long-term secrets are usually used indirectly by deriving from them other secrets for specific applications or communication flows. The value of these assets is tied to both the assets they protect and the opportunities they can enable for an attacker that can access them, most importantly access to the internal networks that the IoT endpoints connect to.

- **Device software**: the software that runs on IoT devices is valuable from two perspectives - it enables the rest of the system, since both previous categories of assets depend on the correct operation of software, and it can be valuable on its own, particularly when it contains important intellectual property. Much like keys and credentials, a sizable part of software's value is indirect, given by its role as an enabler in the system, though the value of trade secrets can also be an important financial incentive, especially with the emergence of small-scale machine learning applications.

2.4.2. Attack vectors

Before determining what types of defence mechanisms to put in place for the assets described in the previous section, it is important to identify the ways in which any of the assets can be exposed or tampered with. [13] splits the security perimeter for the perception layer into two areas: attacks that exploit the network traffic, and attacks that exploit the physical interfaces of the components (such as the MCU chip). [14] also proposes a taxonomy for threats faced by IoT devices, explicitly mentioning attacks involving bus probing and monitoring (described more thoroughly in section 3.1). The European Union Agency for Cybersecurity (ENISA) has also published a report, written with input from industry experts, on Good Practices for Security of IoT [15]. The report goes on to state that:

> *However, securing IoT, and especially IoT edge-devices, can prove a difficult task for software developers if hardware comes without basic security capabilities. For example, when implementing a strong cryptographic algorithm in the software stack, it is the use of a Trusted Platform Module (TPM) in the hardware that will ensure the private key cannot be exposed.*

[15] also endorses the OWASP Top 10 IoT list as a noteworthy resource. The Open Web Application Security Project™(OWASP) publishes security guidance reports for a variety of systems, with their Top 10 lists of common issues being the best known. The OWASP Top 10 IoT [16] list includes two points relevant for this thesis: insecure data transfer and storage ("Lack of encryption or access control of sensitive data anywhere within the ecosystem, including at rest, in transit, or during processing"), and lack of physical hardening ("Lack of physical hardening measures, allowing potential attackers to gain sensitive information that can help in a future remote attack or take local control of the device").

Many of the security issues relevant to IoT devices revolve involve problems with either the software running on them (e.g., the presence of insecure software components that are accessible over a network), with their provisioning or deployment (e.g., the use of insecure default settings), or with vulnerabilities present in certain interfaces (e.g., in the radio-frequency interface). Such vulnerabilities, however, are irrelevant in the context of this project, as their exploitation typically does not depend on having physical access to the device. These attack vectors are thus not taken into consideration.

On the physical attack front a few categories of attacks can be distinguished. First off, being capable of probing inter-component buses and reading all data exchanged between components allows an adversary to exfiltrate all data stored in off-chip memory or storage, any sensor reading, and any actuator request. Given that short-term cryptographic keys are usually stored in clear in memory, recovering them completely breaks down any encryption-based security enforced in network protocols. Securing memory by encrypting it on the fly in software does not offer a solution because the encryption key would have to be stored in memory as well. OWASP identified this issue as Insecure Data Storage and Lack of Physical Hardening.

Chapter **Error! Reference source not found.** looks more closely at the technicalities behind possible a ttack vectors and identifies the vectors that this thesis is concerned with.

2.4.3. Mitigations

The task of ensuring secure and reliable virtual interfaces to IoT devices – for everything from network traffic to secure updates – has seen a lot of research and development. Yet most of these mitigations fall outside the remit of this thesis, as physical attacks remain unhindered by them. [14] groups the mitigations required against physical attacks as tamper resistance – "the desire to maintain these security requirements even when the device falls into the hands of malicious parties, and can be physically or logically probed". [15] recommends software-hardware co-design as the only suitable solution, as tackling them in software becomes intractable under a threat model where the attacker has control over the device.

Very few of the hardware mitigations deployed in practice can act as mitigation against an attacker that can perform even passive physical attacks such as monitoring data crossing the device on data buses. The relevant features will be described in chapter 4, though their applicability to IoT devices has not been investigated thoroughly. An approach that has been used extensively in small-scale devices is that used by smartcards, based on a small MCU with hardened hardware and software implementations, with a high degree of resistance against the whole spectrum of attacks, both logical and physical. While this approach has been successful and new iterations of banking cards can even run Java web servers and communicate over hypertext transfer protocol (HTTP), they are limited in scope and performance compared to high-end IoT devices (e.g., a RaspberryPi).

An important thing to note about hardware solutions to security problems is the risk involved in deploying the feature - a major reason for using a hardware solution is the immutability of the implementation. Attackers cannot bypass or disable verifications, but at the same time there is no way to update or modify them. Whenever errors are found, they are permanent and could, at most, be glossed over with software updates. Software Guard Extension (SGX) - an Intel processor feature meant to protect the sensitive part of an application, and which will be described more thoroughly in chapter 4 - is an example in this sense. Since its first implementation was released, a multitude of vulnerabilities were discovered which are generally unpatchable ([17], [18], [19], [20], [21], [22]). Thus, care must be taken when implementing hardware-based security features to aim for low-complexity implementations.

2.5. Protection mechanism assessment criteria

Designing, implementing, and testing defence mechanisms aligned against the types of attacks presented in the previous section requires an understanding of the trade-offs required. Given that the attacks operate on an interface that is both a bottleneck and on the critical path for the whole device, any intrusive processing could lead to heavy performance penalties in the overall functioning

of the device (as will be shown in section 6.3.4). Indeed, the lack of such mechanisms in the wild comes mostly down to the large cost that they typically imply. In the case of IoT devices this cost is even less palatable – adding a burden to already-constrained devices could be a disastrous choice for the producer of the device, as consumers who have little interest in the security of their platforms choose more performant (but less secure) devices. Security of devices found in the field could thus be improved not just by improving the security-related features of the defence mechanisms, but also by reducing the performance toll a user must pay to use them. The characteristics discussed below are also used in [23], where they are used as metrics for assessing their proposed memory security mechanism intended for embedded systems. Chapter 6 goes into more detail about the exact reasoning behind which metrics are more representative (and why), and into the methodology involved in measuring these characteristics.

- **Performance** – as mentioned previously, performance is particularly important, and can be defined in several ways. A usual metric is given by the speed with which the processor executes some given benchmark, but other figures might be just as important. For example, because the mechanisms usually involve some increase in the amount of data read or written from/to memory, measurements of the total data transferred or of the memory bandwidth consumed could be more appropriate. Which measures are more relevant generally comes down to what kind of applications are of interest. Performance also correlates with energy usage, as a slowdown driven by some mechanism would lead to more time spent in an active state.
- **Memory cost** – as will be explained in more detail in chapter 4, mechanisms protecting from physical probing of the memory bus require the presence of metadata associated with the protected memory. This means a cost in terms of available memory – a section of memory is carved out for the use of the protection mechanism only. In the case of IoT devices, many of which are also constrained in terms of the total memory size available, this can be problematic. The memory cost has a complicated relationship with the various performance metrics mentioned earlier. This relationship will be discussed on a case-by-case basis in relation to the defence mechanism designs presented in chapter 4. These trade-offs can also spill into the next characteristic, implementation footprint.
- **Implementation footprint (area)** – the implementation of the defence mechanism requires some physical space on the integrated circuit hosting the processor. This space host everything from the finite state machines powering the mechanism, the cryptographic accelerators that perform all required computations, caches for storing metadata, and so on. This footprint can cut into the space available for other features (such as hardware accelerators), or alternatively can increase the cost of the CPU, if the whole circuit needs to be scaled up. The extra circuitry will also involve more energy consumption. As will be explained in chapter 6, no explicit measurements will be given for this metric – instead, rough comparisons can be made between various implementations given the differences in the mechanism design.

3. Security threats and objectives

On generic computing platforms both the implementation and state of applications can be found in memory – hence, much of computer security deals with ways in which memory contents and access to it can be abused. A study done by Microsoft found that as many 70% of identified software vulnerabilities are related to memory safety [24]. Common issues such as buffer overflows, heap spraying, return-oriented programming (ROP), all involve misuse of memory from software. One of the most notable vulnerabilities from the past decade, Heartbleed, broke OpenSSL (a prominent cryptographic library) not through poorly implemented or weak cryptographic algorithms, but by allowing an attacker to read from the victim's memory. Similarly, two other outstanding vulnerabilities, Spectre [25] and Meltdown [26], were found in 2018 to affect most CPUs produced in the past 20 years. They allowed malicious applications to bypass protection mechanisms and exfiltrate memory blocks from other processes, including belonging to the operating system.

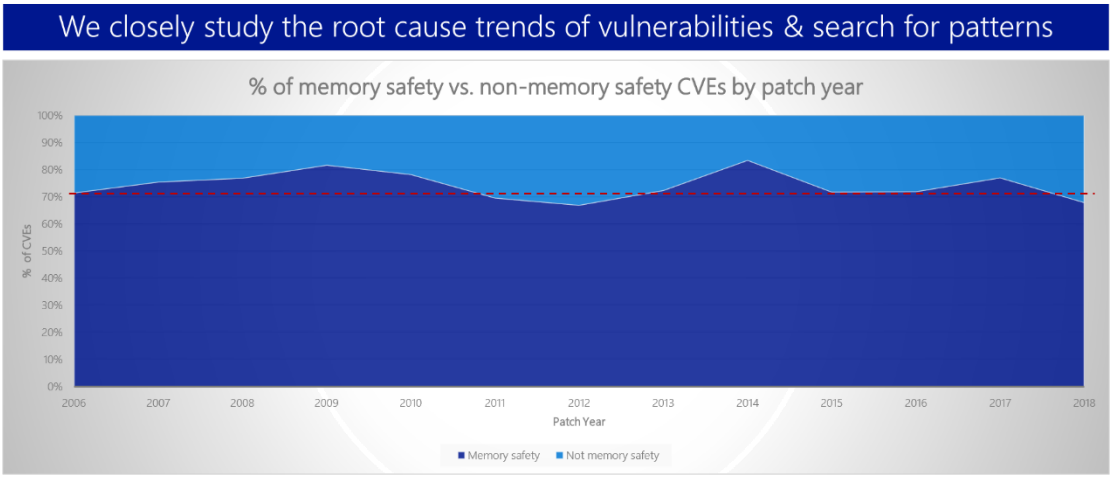

Figure 4 - Memory-related security vulnerabilities continue to form most software issues fixed each year [24]

This thesis, however, aims to tackle some highly specific types of threats which carry significant risks to IoT devices. As discussed in the previous chapter, physical exposure is a common problem for IoT and edge platforms which opens several attack avenues that are both fairly easy to pursue and difficult to mitigate against. This chapter begins with a discussion of the attack avenues tackled in this thesis, as well as a list of attack types that we omit. We continue by setting out our security goals against such attacks along with all the applicable caveats. This is followed by an overview of the types of mitigations that could achieve the security goals we highlighted previously, along with the limitations and applicability of each one. The chapter ends with a discussion of primary memory technologies that pinpoints the ones relevant to our approach.

3.1. Attack avenues

Unfettered physical access to a computing device is a remarkably powerful asset for an attacker. It opens avenues that have historically proven difficult to mitigate against, from various side-channel attacks to fault injection. Out of all these avenues, we are primarily concerned with an attacker's new capabilities around extracting and altering data found in the primary memory of the device. The following sections discuss the types of attacks that are both in- and out-of-scope for this thesis.

3.1.1. Memory bus probing

Memory bus probing involves eavesdropping on the communication bus between CPU and RAM (particularly when packaged separately on a PCB). On-board probing is aided by the fact that in many cases the communication tracks and test points are easily accessible (Figure 5). Even with passive countermeasures deployed - shorter tracks, increased bus frequency, routing of critical buses through the inner layers of the PCB - the strength of the attack can easily outweigh the challenges. An attack example comes from Markus Kuhn who in 1998 defeated the defences of a microcontroller with encrypted memory bus using a personal computer and a US$300 device built from off-the-shelf components [27]. In principle, such an attack would immediately give access to the entire memory space of the device, leaking everything from sensor data to proprietary software, to cryptographic keys used in network protocols.

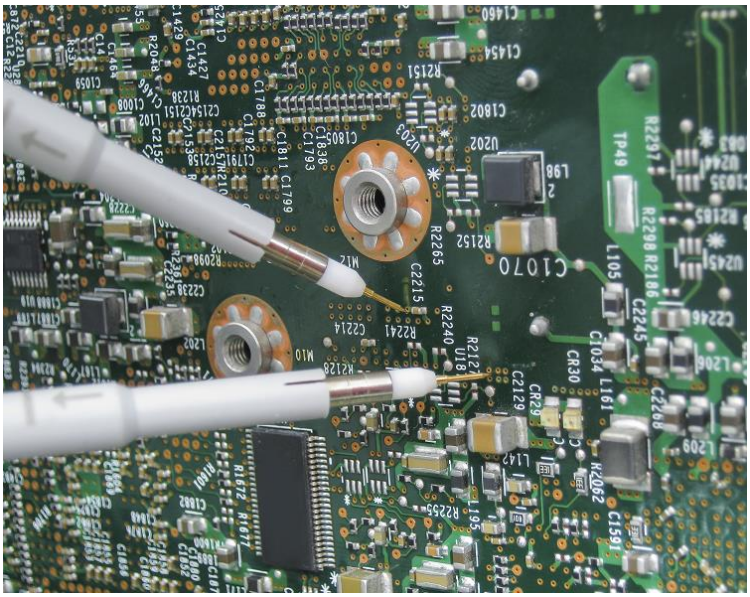

Figure 5 - Physical probing of exposed pins and traces on a PCB [28]

At the same time, the probe can be used to alter the signals being passed, thus acting as an active component. Data retrieved or written during memory transactions can be modified by the attacker, or some operations can be skipped altogether. An example of such a system can be seen in [29], where a rogue memory controller connected to exposed DIMM socket pins was used to exfiltrate and modify data found in main memory.

3.1.2. Memory bank manipulation

Instead of targeting the bus which connects the processor to memory, the attacker could, instead, modify the physical setup of the memory chip. One option is the insertion of an interposer between the RAM chip and the PCB (Figure 6). The interposer bridges the chip to the PCB while providing easily accessible pins. The attacker can then use these pins to perform attacks in a similar way to the bus probing described above. Such a setup was used to perform the Membuster [22] attack against the memory protection offered by Intel SGX.

An alternative approach involves a specialized replacement to the whole memory chip. In essence, the whole interposer-memory chip ensemble described above would be replaced by one single component which acts as a software-defined memory chip, with the attacker controlling all transactions, modifying those that could aid in the attack and extracting all relevant data.

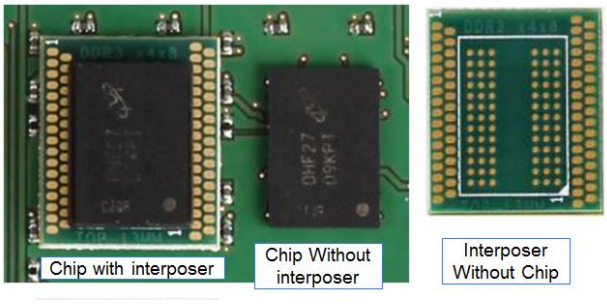

Figure 6 - Use of a DDR3 chip interposer [30]

3.1.3. Cold boot attacks

Another category of attacks targeting memory extraction from a running system is that of cold boot attacks, where the RAM chips (either SRAM [31] or DRAM [32] [33]) are physically removed from the victim device and connected to a separate device controlled by the attacker. These attacks work because, despite RAM being usually volatile, the contents can be preserved for some amount of time, depending on the underlying technology. Cooling down the memory chip helps slow down the degradation of data in the chip. Thus, if the switch is performed quickly enough after the chip has been brought to a low temperature the whole memory space becomes accessible to read. Figure 7 shows the degradation of data found in memory following such an attack, with respect to the time taken to reconnect the memory chip to a power source. It is, thus, feasible to extract large amounts of information with relative ease using a cold boot attack.

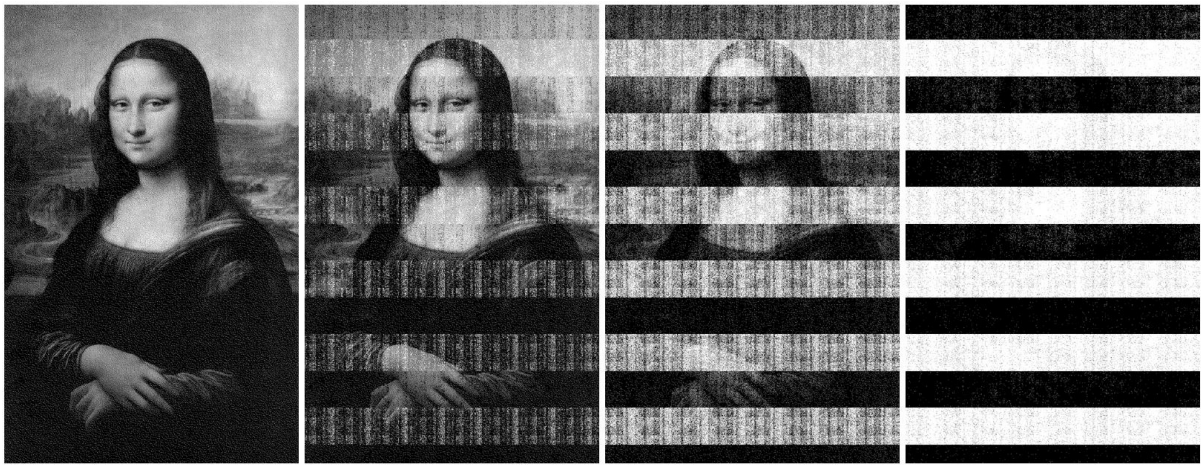

Figure 7 - Stages of degradation of an image in memory depending on time spent with no power provided to the memory bank (from left to right: 5s, 30s, 60s, 300s) [33]

3.1.4. DMA attacks

A less invasive type of attack that has the same goals – reading and writing directly from/to main memory – can be achieved by legitimately connecting a peripheral device to the system and issuing Direct Memory Access (DMA) commands to read or write from/to physical memory. An attack example is TRESOR-HUNT [34], which makes use of DMA reads and writes to extract a (presumably) CPU-bound secret. A practical setup for another such attack can be seen in Figure 8, which was used to exfiltrate the FileVault encryption key straight from the memory of a laptop [35].

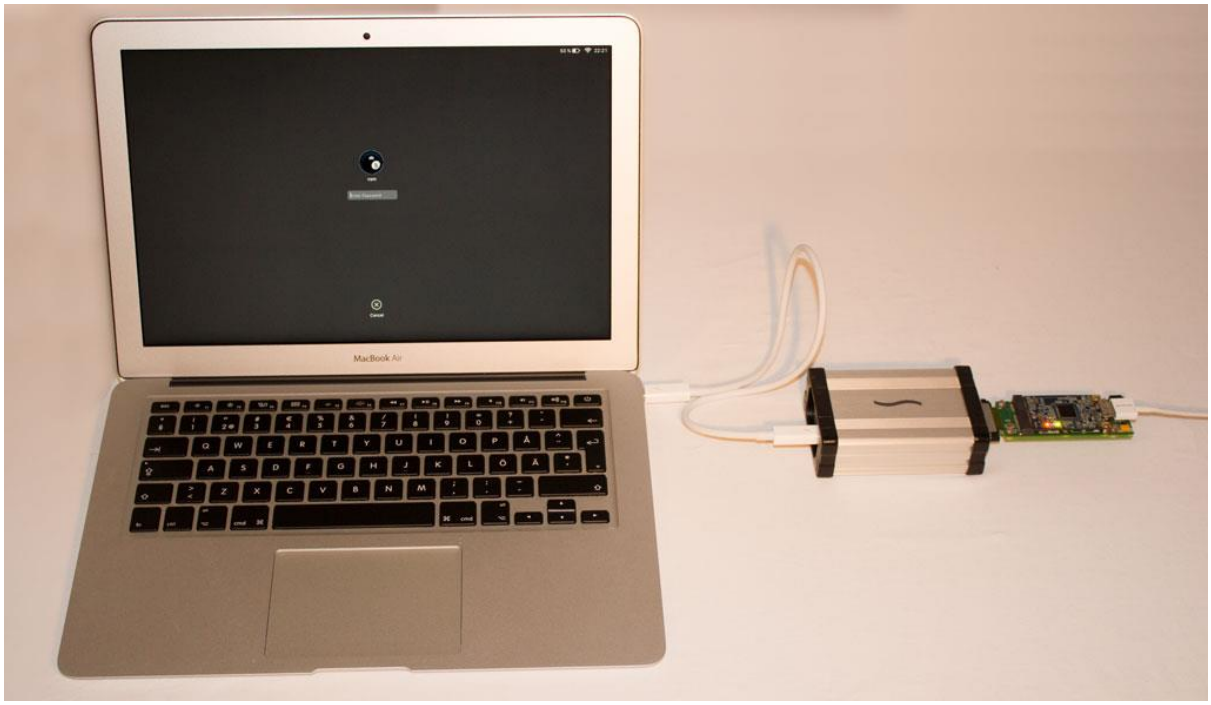

Figure 8 - Laptop with PCILeech hardware connected for a DMA attack [35]

3.1.5. Out-of-scope attacks

The attacks outlined above represent only a sliver of the whole attack surface offered by an IoT device. The following sections give a high-level overview of the types of attacks that fall outside the remit of this thesis. The main reason for ignoring these attack avenues is that the mitigations needed to defend against them are merely complementary to the mechanisms needed to defend against the attacks discussed earlier in this section.

3.1.5.1. Software-based attacks

Given the growing complexity of software stacks deployed even to modest computing platforms, and the statistics regarding memory-related security vulnerabilities in software (Figure 4), the logical interfaces of every IoT device can be expected to contain at least some bugs ripe for exploitation. These could range anywhere from classical stack overflows to Rowhammer attacks [36]. Regardless of whether these vulnerabilities are triggered by data sent to the device, or by malicious software that is allowed to run on the device (including by privileged software, such as in Iago attacks [37]), the attacks fall outside the scope of this thesis.

3.1.5.2. Glitching and fault-injection

Glitching and fault-injection attacks rely on physical access to control the operation of a device in a way that was not intended by the manufacturer or owner. These attacks have diverse goals (e.g., skipping instructions [38], or leaking cryptographic secrets [39] [21]) and can be achieved in a multitude of ways (e.g., using electromagnetic pulses [38], or underpowering the device [40]). Despite their status as physical attacks, they are nonetheless not included in our scope.

3.1.5.3. Micro-probing

A particularly invasive alternative to the probing technique described in section 3.1.1 is micro-probing, in which a small probe is connected to the active circuit of the processor chip, for example

to a data bus, to acquire a desired signal (Figure 9) [41]. These intra-chip probing techniques, however, require special expertise and equipment, must be performed in a laboratory, and can heavily damage or destroy the chip. Their cost and difficulty put them outside of our remit.

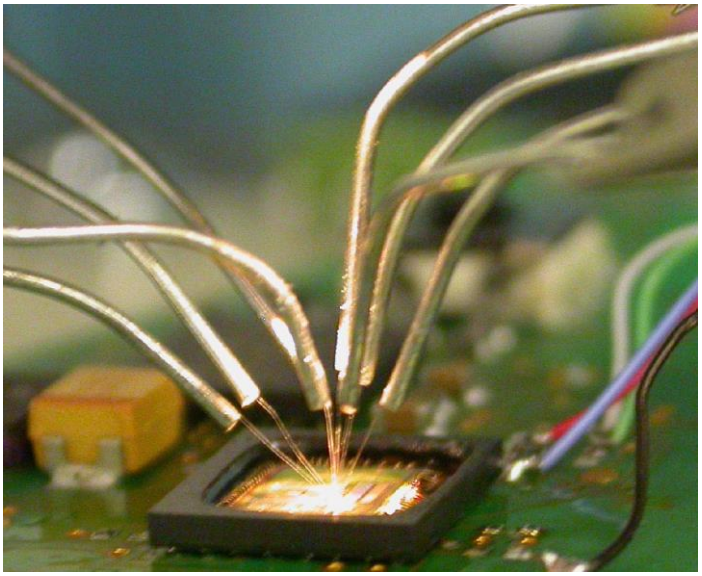

Figure 9 - Micro-probing a chip; fine probes are dropped on the decapped IC, allowing the acquisition of signals travelling within the circuit [41]

3.1.5.4. Other side-channel and speculative execution attacks

Side-channel attacks typically involve unconventional use of a mechanism – be it software or hardware – to extract information that is not normally accessible. Information is generally leaked by a less-than-perfect implementation of the mechanism, through a medium that is orthogonal to the actual purpose of the mechanism [42]. The organisation of the memory system can, for example, be used to yield information about memory layout of privileged processes using timing side channels [43]. While many of the hardware components involved in our prototypes have been the subject of side-channel attacks in the past (e.g., caches [44] [45] [46]), we consider these attacks relevant only to hardware prototypes or implementations and thus outside of our scope.

Another side-channel worth mentioning that is also beyond our scope relates to the memory transaction metadata. This metadata can help an attacker identify fine-grained access patterns of target processes, which can ultimately be used to extract sensitive information [22].

3.2. Security goals

Now that the relevant attack avenues have been listed, the desirable security goals can be stated in relation to the attacks.

- **Data confidentiality**: confidentiality is required to mitigate against attackers reading data from the main memory using one of the physical attacks highlighted in section 3.1. The attackers should either be unable to obtain the data using the attacks found within our model or should be incapable of extracting any meaningful information from this data, even when given generous amounts of time, computation power, and storage space.
- **Data integrity**: integrity is required against the same attackers altering in any way the data stored by the processor. Like for confidentiality, a strong guarantee (in the negative) is preferable when it comes to the likelihood that the attacker could successfully have altered

data accepted by the system. However, in some situations this guarantee can be relaxed to act only as a deterrent.

Despite the seemingly straight-forward nature of the goals, several caveats exist. The first is that the defence mechanisms are expected to fulfil the goals stated above only in relation to data stored at runtime in primary memory. Data persisted across reboots in secondary memory is outside the scope of this project.

The second caveat is that we are concerned with the (indiscriminate) application of the goals above to all data stored in memory. Management of the exact security guarantees necessary for each memory section falls beyond our goals, as an extra feature of the security mechanism.

3.3. Mitigation taxonomy

Multiple approaches to ensure the goals we set out above are technically possible. This thesis focuses on the approach highlighted in section 3.3.4, however we offer a brief overview of other techniques in the following subsections.

3.3.1. Tamper-resistant design

The most generic type of mitigations against the attacks described above is to put in place sufficient physical defences to make the attacks too costly or difficult [47]. Tamper-resistance against physical attacks usually relies on active and/or passive measures intended to either prevent or detect attacks (for example by detecting physical penetrations of the protection layers, or unusual voltage fluctuations in the device), and to respond accordingly [48]. This approach has been perfected in the field of smartcard development, with research going towards developing generic tamper-resistant computing devices [49] [50] [51]. Clean-up of sensitive memory areas as a result of tamper detection is also a research field on its own [52]. While these techniques are effective as a deterrent, they also increase the cost of the device [48].

3.3.2. Smart memory bus encryption

An alternative mitigation approach is the use of a secure communication channel between processor and memory. This requires a memory chip with processing capabilities [53] that could participate in a (mutually authenticated) cryptographic protocol which ensures the confidentiality and integrity of all data exchanged with the processor. These techniques could successfully mitigate against all the proposed attacks, with the lower bound for memory exfiltration/alteration set by the physical security characteristics of the RAM chip. STASH [54] has already been proposed for compute-capable hybrid memories (mixing DRAM and NVRAM banks) which factor physical attacks into their threat model. This design, however, fulfils all the goals that were put forward in section 3.2 not only with the use of memory bus authentication, but also with techniques as described in 3.3.4. InvisiMem [55], another similar design, only provides authenticated encryption over the memory bus, avoiding the techniques mentioned in 3.3.4 and relying, instead, on remote attestation between the processor and memory chips to bootstrap trust between them.

A drawback of these technologies is the extra complexity involved in having new hardware and/or software components added to both the processor and the memory chip to handle these new mechanisms.

3.3.3. Homomorphic encryption

A completely different approach is the use of homomorphic encryption (HE) with the sensitive data being processed. HE allows encrypted data to be processed without having to decrypt during the actual processing steps and is thus suited for computation done in untrusted environments such as cloud servers or IoT devices. Fully homomorphic encryption (FHE) allows arbitrary processing of the encrypted data to obtain some sought-out result that is, nonetheless, itself encrypted. Though FHE schemes have only been known for little more than a decade, constant progress is being made in perfecting them [56] [57] [58]. Data processed under such a scheme on an IoT device could be exfiltrated through one of the attacks in 3.1, but not decrypted by the attackers. However, if no other mitigations exist, attackers could still take control of the device.

FHE is still problematic in terms of speed [59] and applicable only where no sensitive data exists unencrypted on the device.

3.3.4. Cryptographic memory protection

Cryptographic memory protection is yet another alternative and the main topic of this thesis. Much like the approach discussed in 3.3.2, it uses cryptographic techniques to enforce our desired goals. However, unlike in the case of smart memories, the only trusted component in this model is the processor, whose task it is to secure and verify all outgoing and incoming data. The mitigation plan involves encrypting all data sent to memory and ensuring its authenticity via authentication tags. A more in-depth analysis of the algorithms involved, their properties and role, and an overview of optimizations proposed so far can be found in chapter 4.

Our motivation for the interest in this approach (as opposed to the ones presented above) is multifaceted. An approach to physical memory security that relies on features of the processor alone is easier to deploy (and likely cheaper) than one requiring multiple components to orchestrate the defence. It can also protect all data in memory, including application code and any transient secrets, not just data that was encrypted by an external actor and sent encrypted to the device. Overall, we believe this approach achieves the best security guarantees for the lowest complexity.

As will be shown in chapter 4, much progress has been made towards bringing the performance of cryptographic memory protection to an acceptable level for deployment in the general IoT market. With the results from this thesis, we hope to continue this process.

3.4. Relevant memory technologies

The attacks and goals presented thus far relate directly to the security of primary memory in computing systems – however, not all the attacks apply consistently to all types of main memory deployed in IoT platforms. For example, memory bus probing or cold boot attacks are only feasible when the memory bus or chip are physically exposed, whereas DMA attacks only require that new peripherals are logically connected to the memory bus.

External memory chips, usually in the form of SRAM or DRAM memory, are the most prone to physical attacks. Apart from DMA attacks, all other avenues in our scope require the physical exposure that such a setup offers. At the same time, external memory is common among the high-end devices discussed in section 2.3, either as DDR or Low Power DDR (LPDDR) modules [9].

Another option for primary memory is on-chip memory (i.e., memory integrated on the same chip as the processor), such as SRAM blocks residing on the same substrate as the CPU, package-on-package

DRAM (PoP DRAM), or using though silicon via (TSV) technologies such as High Bandwidth Memory (HBM) [60] or Hybrid Memory Cube (HMC) [61]. The offensive avenues equivalent to probing and interposing attacks on external memory are found in micro-probing attacks as described in section 3.1.5.3, which are nonetheless outside of our remit. Such memories are thus only relevant as far as DMA attacks are possible by rogue peripherals.

A third option consists of non-volatile RAM (NVRAM) – a type of memory whose contents can be recovered after power loss to the device. This feature is particularly alluring for IoT devices which must operate in environments with inconsistent access to power. However, NVRAM technologies present some characteristics which put them outside the scope of our project. First off, their requirement for system recovery in the face of a crash creates the need for continuous consistency of all data stored in memory, including cryptographic metadata. Memory update operations must therefore be atomic and tailored for NVRAM [62]. Secondly, typical approaches for memory encryption designed for DRAM technologies (which rely on encryption's diffusion property) can result in wear-out of NVRAM chips and very poor performance overall [63]. While plenty of research has brought massive improvements to the security guarantees of such memories [62] [63] [64] [54], these special characteristics demand a bespoke approach to the security requirements. We have, however, decided to focus our attention on protecting DRAM technologies, as this is the most prevalent form of primary memory found in current mid- and high-end IoT devices [9].

4. Cryptographic protection mechanisms

Following on from the overview on physical memory security given in the previous chapter, we continue with an in-depth discussion of our mitigation of choice against physical attacks – cryptographic memory protection. Before diving into the technical details, we begin by mapping the goals defined in section 3.2 to the interaction between processor and memory.

From a simplistic point of view, the execution of a processor can be modelled as communication or data exchange between the processor and itself at different points in time: value A is placed into runtime memory only to be retrieved at a later point and used in some way. Memory can thus be considered a communication medium. Data stored in memory could contain sensitive information with sufficient value to motivate a third party into extracting or modifying it.

If we want any malicious third party to be incapable of accessing the data we store in memory, we must ensure its *confidentiality*. If we want to be sure that data found in memory was indeed written by us, we must ensure its *integrity*. Devices found under strict physical control (such as personal computers) already have an acceptable solution for both of those services in the physical restrictions present in their environment. IoT devices deployed in uncontrolled environments, on the other hand, do not benefit from physical or administrative controls and must therefore rely on technical controls built within the device. The universal solution for internet traffic (including for IoT devices) that demands the same security services is to cryptographically protect it.

The exact boundary beyond which data is considered untrusted must also be established. Assessing the security of the system relies on a clear definition of this boundary. Data flows can then be tracked to make sure that processing and checking is done at the correct points. Figure 10 below shows the placement of the Memory Protection Engine (MPE) that forms the subject of this project. Confidentiality and integrity are needed specifically against an attacker that can probe the memory bus; thus, we can set the trust boundary at the physical edge of the processor chip. The logical boundary is then in the component that mediates between the two domains – the memory controller which translates between requests issued by the processor and the memory bus protocol. A more comprehensive description of the system operation will be given in chapter 6.

This chapter starts with a discussion of the high- and low-level primitives needed to fully achieve the goals introduced in section 3.2. It continues with a review of the performance improvements proposed and/or implemented thus far, aimed at making such techniques more palatable for device manufacturers. This review also covers the applicability of these techniques to our own prototype.

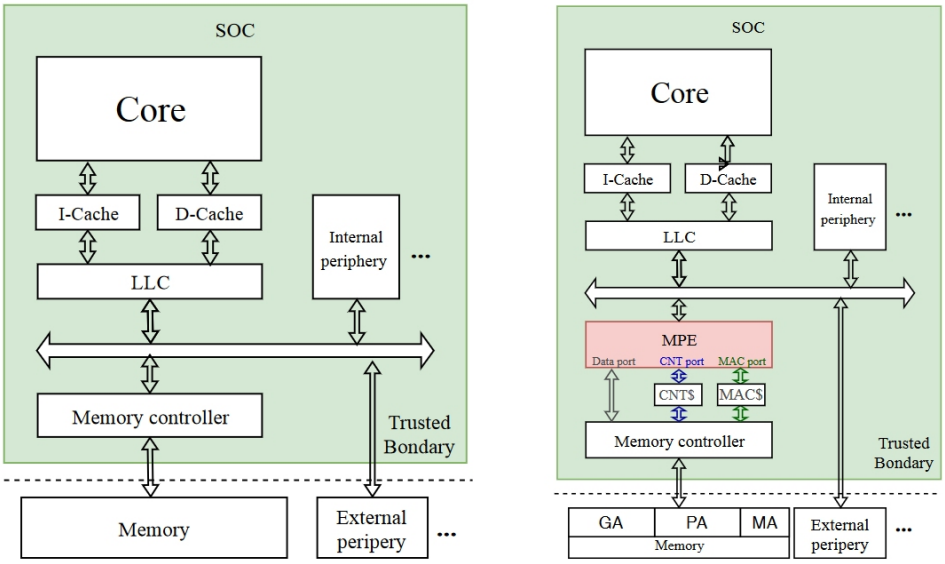

Figure 10 - MPE positioning within a processor [4]

4.1. Basic mechanisms

4.1.1. Confidentiality protection

Confidentiality is a property indicating the inability of unauthorized entities to access the underlying data, or to extract meaningful information about it. Confidentiality is generally provided by encryption algorithms in cases where unauthorized entities cannot be prevented from intercepting the processed data. Encryption works by taking the original data (called plaintext from now on), applying to it a transformation parameterised by a secret key, and sending the result (called ciphertext from now on) to the other party (in our case, to memory). When the ciphertext must be read, the opposite transformation – which is also parameterised by a key – must be applied.

Encryption algorithms have a diverse taxonomy. Two broad categories are represented by symmetric and asymmetric algorithms, with the high-level difference being in the key usage pattern. For asymmetric primitives encryption is performed with a “public” key while decryption is performed with a “private” key. Such constructions are useful in transmitting data between two entities who do not already share an established connection. The only person capable of decryption is the owner of the private key and the public key can thus be shared publicly – hence the public/private naming for keys and the asymmetry between their usage. Symmetric encryption, on the other hand, typically uses the same key value for both encryption and decryption and thus requires a pre-established key before confidential data exchange can happen. Given that the MPE will perform both encryption and decryption (and thus does not need key establishment between two parties) and given the large performance discrepancy between asymmetric and symmetric algorithms (in favour of the latter), symmetric encryption is the natural choice for our use case.

Symmetric primitives can also be split quite broadly into two categories, stream ciphers and block ciphers. Stream ciphers handle arbitrary amounts of data per operation while block ciphers process data in fixed-size blocks (typically 128 bits). The flexibility of stream ciphers proves helpful in situations where sporadic bursts of data must be encrypted efficiently.

The security of block ciphers is usually tied to their key size since for a well-designed cipher the difficulty of reversing the transformation (i.e. obtaining the plaintext corresponding to a ciphertext) is equivalent to identifying the key with which the transformation was done. Determining which ciphers are well designed, however, has been a global undertaking in the past few decades. This process has improved both the levels of confidence in a multitude of ciphers, and the understanding of what makes a strong cipher.

The quintessential block cipher is the Advanced Encryption Standard (AES) [65], chosen in 2001 following a competition organised by the National Institute of Standards and Technology (NIST). AES works on plaintext blocks of 128 bits using keys 128, 192 or 256 bits long, and consists of a number of processing rounds (either 10, 12, or 14, depending on the key size). Performance-wise it presents a balanced design that makes it reasonably well suited for a large field of uses while offering a high level of protection. AES has been deployed in network protocols (e.g. TLS), hard-disk encryption products (e.g. BitLocker) [66], cryptographic memory protection (AMD's Secure Memory Encryption [67], Intel's SGX [68]), and many more applications.

4.1.1.1. Modes of operation

Despite being strong primitives on their own, the way in which block ciphers are used to encrypt in real-world applications is just as important for the confidentiality of the encrypted data. Modes of operation describe this process that links plaintext, cipher, and other metadata used to strengthen the construction to create a ciphertext. To understand why modes of operation are needed it is enough to look at the most basic mode: Electronic Code Book (ECB). In ECB, the cipher is used directly to encrypt every 128 bits of data as shown in Figure 11.

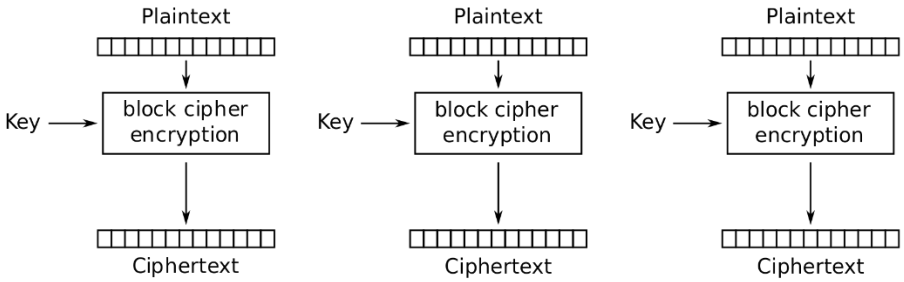

Figure 11 - Electronic Codebook mode, i.e. direct encryption of the plaintext under the given key

This simplistic approach has a big drawback which makes its use strongly discouraged: if two plaintext blocks are identical, the resulting ciphertext will also be identical. This can end up leaking a large amount of information about the plaintext, as can be seen in Figure 12. The image of Tux on the left is shown encrypted under ECB (middle) and under a secure mode of operation (on the right).

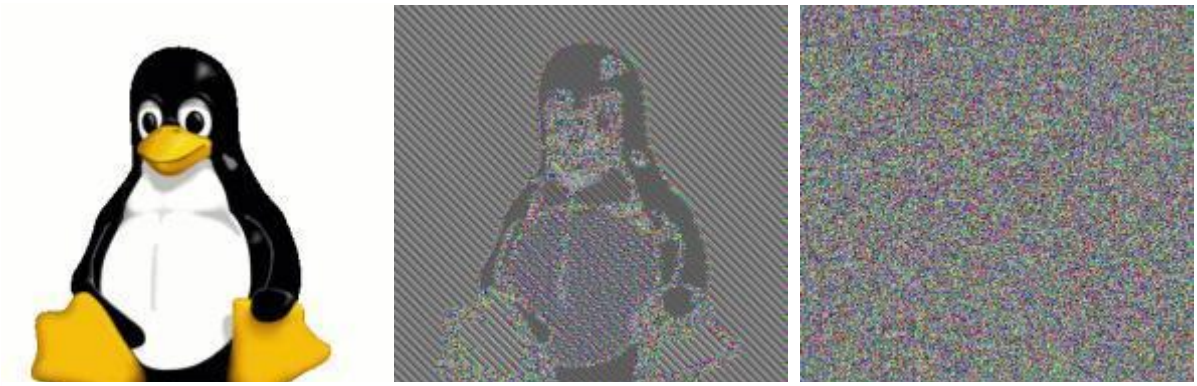

Figure 12 - Information leakage about the plaintext due to ECB mode encryption ([121], [114], [113])

A multitude of operation modes exist, each with different characteristics that make it more suitable for specific applications. Efficient and secure memory encryption depends on two features that can be brought by operation modes:

a) **Mixing in of location metadata**: when a memory block is encrypted, the ciphertext must depend not only on the plaintext and the secret key, but also on the block's location in memory. Most modes of operation include by default some temporal metadata (such as a counter or random nonce) which must never repeat and mitigates against the weakness described above for ECB – since that metadata changes for each encryption, the same plaintext is always encrypted to a different ciphertext. Adding location metadata (e.g., by using the physical address of the memory block) adds another dimension to the protection, allowing different memory blocks to share temporal metadata without a loss of confidentiality. In a sense, each memory block gets its own parameterized encryption function. Thus, encrypting the same block of plaintext for storage at two different locations in memory will produce different values even if the encryption key and temporal metadata used are identical.

b) **A short critical path**: as mentioned in section 2.5, the performance impact of cryptographic memory protection is the main variable to optimize for. This is a critical goal because the processing done by the new hardware module is directly on the memory access path. Any data brought in from memory must be decrypted and integrity verified before it can be released to the processor. Ideally, the processing would be instant, and the latency would not be increased beyond the time needed to bring the data from memory. The next best option is a protection mechanism with a short critical path. A more thorough description is given in section 4.2.3.1 for how to obtain an improvement to critical path length for memory encryption by relying on the parallelism that can be achieved by memory requests. Operation modes or ciphers that allow parallel computations and/or precomputation of encryption tags are thus particularly well suited for this.

Feature (a) described above effectively mandates that the mode of operation can take and mix in metadata to be used for differentiating between data encrypted at different locations in memory – not all modes of encryption provide such a feature. ECB, for example, offers no way to incorporate such metadata. Counter mode (CTR), on the other hand, offers a facile way for incorporating metadata, described more thoroughly in section 4.2.3.1. Other modes, such as XTS [66] or ACME [64], have similar properties, but fall short due to being optimized for different applications – XTS for hard-disk encryption and ACME for non-volatile RAM. The flexibility specific to modes of operation can fulfil the requirements listed above but cannot overcome all the shortcomings of the underlying

block ciphers. Counter mode can bring the critical path down close to a simple XOR but it cannot reduce the tag generation time or the implementation size of the algorithm below a minimum specific to, say, AES. It is, therefore, more efficient to design the block cipher from the ground up around its intended use-case. The last two decades of block cipher proposals and research aimed at breaking them has made tailoring new ones more accessible. Section 4.2.1 goes into more detail about the primitives tailored for the type of use-cases under which cryptographic memory protection can be placed.

4.1.2. Integrity protection

Integrity is a property indicating the inability of unauthorized entities to modify the underlying data without authorized users noticing. Integrity is necessary even when confidentiality is already ensured through encryption because attackers could modify the ciphertext in a way that introduces a controlled change in the plaintext. ("Malleability" is the property of block ciphers and modes of operation that describes this issue.) Even though in some cases the integrity of data can be verified in an abstract way in software after decryption (e.g., by checking if the result of decryption can be interpreted as an English sentence encoded in ASCII characters) in many cases this is not feasible. This is especially true when the plaintext is supposed to be entirely random such as in the case of cryptographic keys.

Integrity of data can be ensured through a cryptographic mechanism where a "fingerprint" of the plaintext or of the ciphertext is created and transmitted along with the ciphertext. The fingerprint depends on all the input data (and potentially on some relevant metadata) and on a secret key which allows the communicating parties to verify its validity.

A fingerprint can be understood as a condensed, fixed-length version of an arbitrarily long message. Memory chips tend to include a fingerprinting mechanism called Error Correcting Codes (ECC), intended to protect against and to help recover from accidental changes to its contents. ECCs, however, are not parameterized by a key and can be recomputed and replaced by an attacker who changes memory contents. Similarly, cryptographic hash functions are mechanisms that map messages of any length to a short value (say, 256 bits) called a digest. Cryptographic hash functions satisfy several properties that make it difficult for an attacker to reverse the computation and identify a message corresponding to a given digest. Calculating the digest of a given message is, however, straight forward and thus an attacker modifying data in memory could also modify its digest if that were the only integrity protection available (with the caveat that if the digest is computed on the plaintext but only the ciphertext can be modified, the computation becomes more intractable).

Message Authentication Codes (MAC) are the cryptographically secure way of ensuring the integrity of data through symmetric-key algorithms. MACs typically involve a digest that either includes a secret key during computation or is encrypted with a symmetric algorithm. Such an encryption can happen either directly or using an encryption pad, as described in section 4.1.1.1. However, when using an encryption pad the problem of malleability becomes more pressing. If a generic hash is protected in this way, an attacker could modify the ciphertext and "substitute" the hash of the ciphertext with the old one with the simple observation that XOR is a self-inverse operation. Thus, if the tag is $T = H \oplus E_P$ (where H is the hash and E_P is the encryption pad), the attacker could recompute the new hash H′ and update the tag to $T' = T \oplus H \oplus H' = H \oplus H \oplus H' \oplus E_P = H' \oplus E_P$. The solution is the use of hash functions that are parameterized by a secret key, which could thus not be computed by an attacker. Universal Hash Functions (UHFs) [74] are an example of hash families used

for this purpose. Just as in the case of confidentiality, constructing integrity tags in this way allows for latency to be parallelised away, as described in section **Error! Reference source not found.**.

Protecting the integrity of memory contents by using MACs is necessary, but not sufficient. Due to the nature of the threat, two specific types of attacks – splicing and replay attacks – need more complex mechanisms than a simple integrity tag. These attacks are discussed in the following section.

4.1.2.1. Splicing and replay attacks

Splicing attacks occur when the memory contents of one block is replaced with the contents of another (Figure 13). Under a system that protects only the confidentiality of data this attack could succeed when no location-metadata is used for encryption. The case is similar even when integrity protection is in place – the attacker can simply switch both the memory blocks and their integrity tags, and the system would be unable to detect the attack. When location metadata is mixed in in a suitable way, splicing attacks can no longer succeed since data in memory becomes locked into a specific location. In the case of an attack, even if incorrectly decrypted data does not lead to software failures, integrity tag verification will fail, notifying the system of suspect activity.

An example of a splicing attack could involve the attacker switching the contents of an area known to contain a cryptographic key with another that contains a known or easily guessable value. All cryptographic operations with that key would then be compromised.

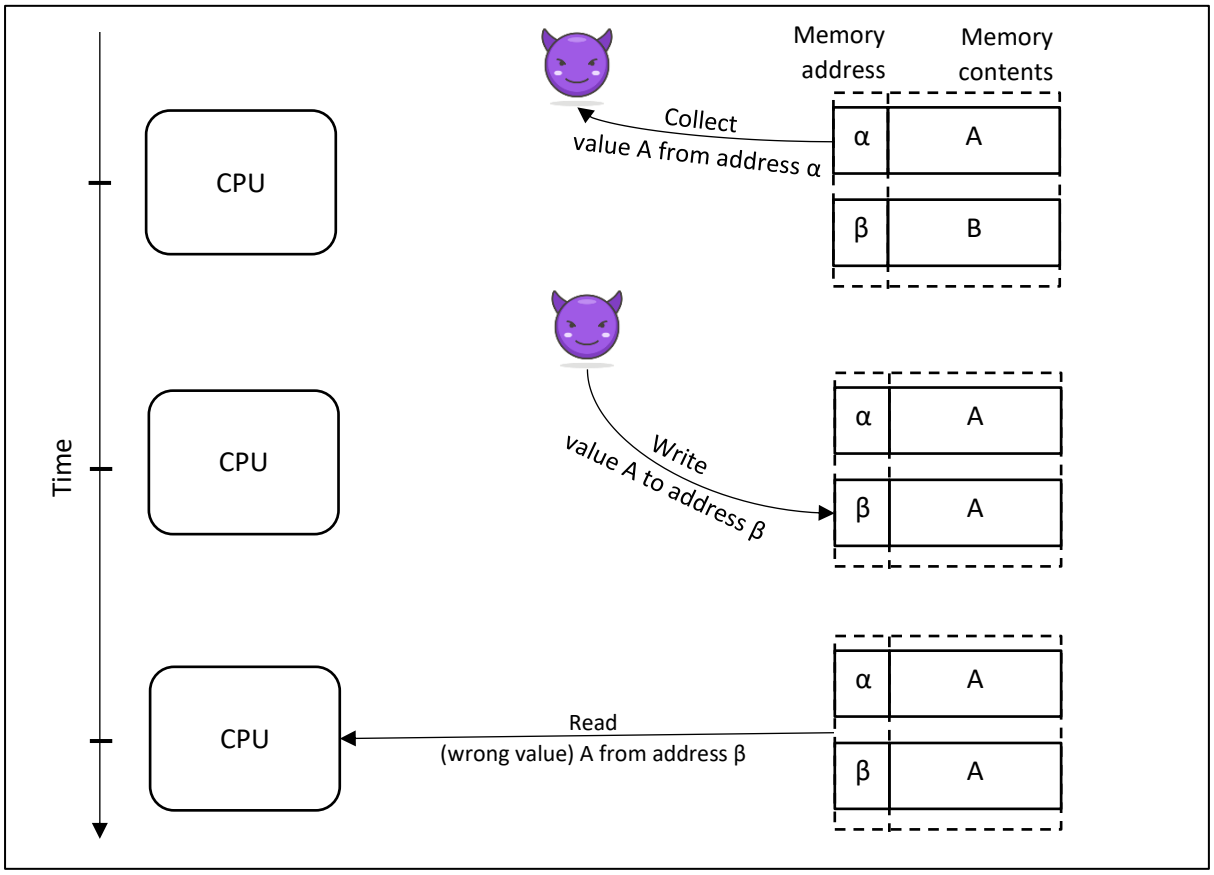

Figure 13 - High-level diagram of memory splicing attack

Replay attacks occur when the contents of a block (and its metadata) are swapped with an older version of the same block. Figure 14 shows the dynamics behind the attack. The attacker starts by

collecting the contents of a memory block at a time of their choosing. They then wait for some section of memory to be replaced with a newer version by the processor. When the processor attempts to read the new value, the attacker interferes and replays the old version.

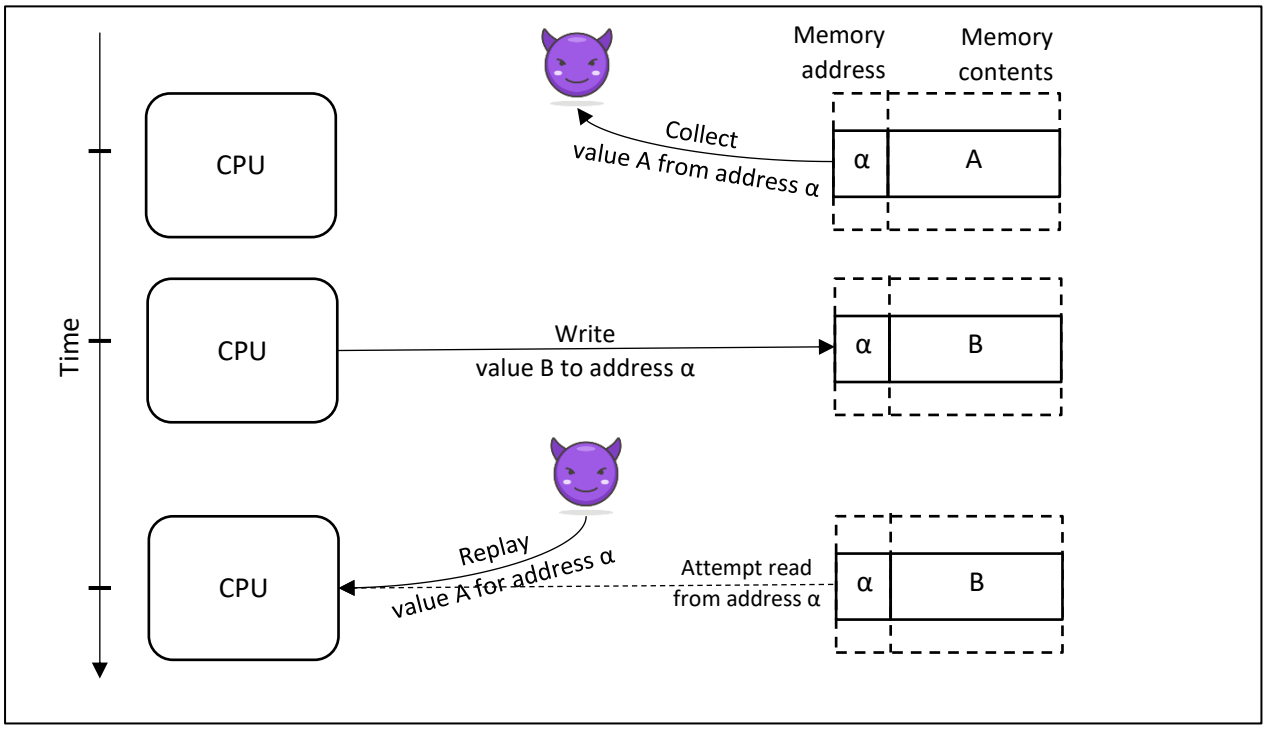

Figure 14 - High-level diagram of a memory replay attack

The processor must be able to identify the old value as a forgery. Nonces (and other temporal metadata) are used in authenticated encryption for this purpose, differentiating between different computations using the same algorithm. Regardless of whether the nonce is generated randomly or through a deterministic process (e.g., a plain counter), it is transmitted/stored alongside the ciphertext and integrity tag, allowing the receiving entity to validate its uniqueness. In the case of two entities communicating, both can track the uniqueness of such nonces and raise an alarm when a duplicate appears. For memory encryption, however, there is no straight forward way of ensuring such uniqueness. If the nonce is stored in memory along with the integrity tag, the attacker may simply copy and replay it as with the rest of the data. Storing it within the CPU would normally use up too many resources on the chip. The next section discusses methods for bootstrapping trust in a large section of memory starting from a small, on-chip register in the processor. Our goal is to find an efficient way of ensuring that no replay attacks can occur.

4.1.3. Replay protection

As was discussed at the end of the previous subsection, protection against replay attacks cannot be ensured through values stored in memory alone, as any value leaving the processor chip can also be replayed. Unlike spatial metadata (in the form of memory addresses used by the processor) which is static and can be relied upon when verifying the integrity of data against splicing attacks, there is no reliable source of temporal metadata that the processor can verify. Security must therefore be bootstrapped from within the processor chip through some other means. Merkle trees fulfil these requirements – they allow the verification of a large amount of data using a small amount of data that can be stored within the chip.

4.1.3.1. Merkle trees

The purpose of Merkle trees is to allow easy verification and updating of large chunks of data. This is done by constructing a binary tree from hash values, where the leaf nodes represent cryptographic hashes of data blocks and parent nodes are simply the hash of their children (as shown in Figure 15). The root of the tree (called Top Hash in the figure) thus depends indirectly on all the data blocks covered by leaf nodes and, by way of the avalanche effect of cryptographic hash functions, a small change in any of the blocks could be detected. The verification process starts at the data block level and continues upward towards the root node. For example, to verify the integrity of block L2, its hash is computed and compared with the value of tree node 0-1. The integrity of node 0-1 is then verified by comparing the computed the hash of nodes 0-0 and 0-1 concatenated with node 0. Nodes 0 and 1 are then concatenated and hashed and the result compared with the Top Hash. If any of the comparisons produces a mismatch, the process is stopped, and an integrity violation is flagged. While storing the whole tree would incur a massive resource burden on the processor chip, storing just the root node is feasible and provides the same level of protection. Even if an attacker can modify any of the data blocks and to recompute the Merkle tree nodes stored in memory, the final verification which involves the node stored on-chip (and which we assume the attacker cannot modify) would fail. All nodes that are part of a tree used to preserve the integrity of data in memory is henceforth called an Integrity Node (IN).

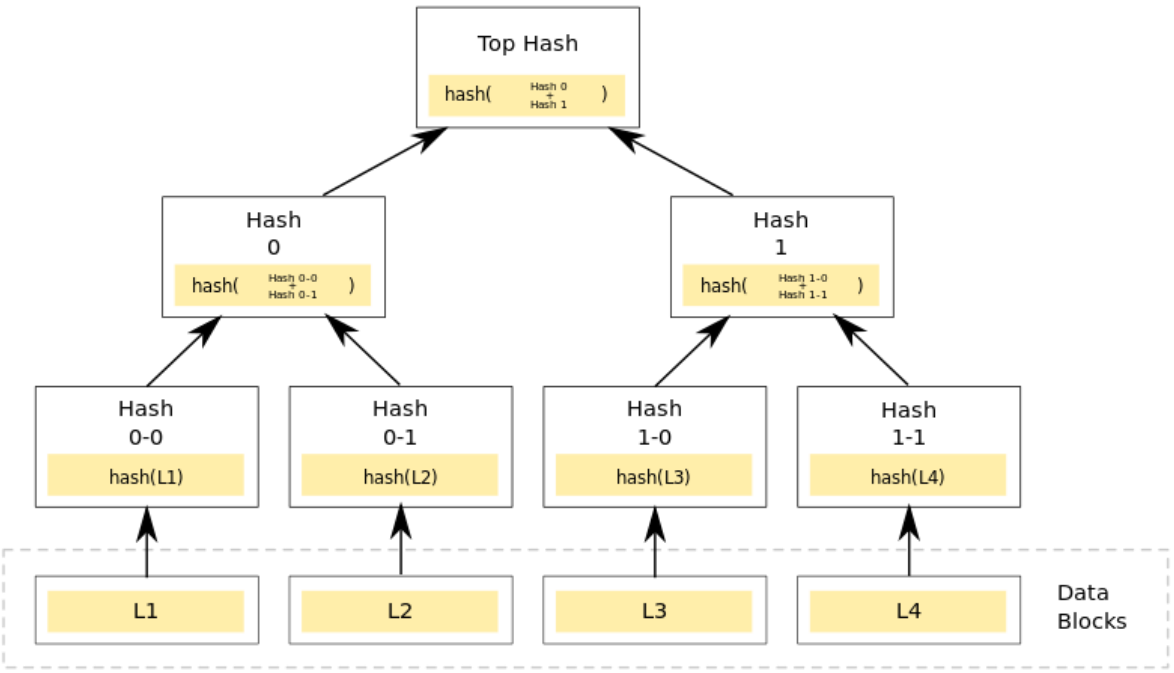

Figure 15 - Merkle tree built on top of 4 blocks of data

Merkle trees can thus ensure the integrity of data stored in memory, including against splicing and replay attacks. This is because no modification the attacker can perform can change or roll back the root hash. Verification will identify any integrity violation within the limits of the hash function used for constructing the tree. If a reasonably secure function is used (e.g., SHA256), the system should be secure enough for the foreseeable future. They were thus the suggested replay protection mechanism in a number of early designs [77] [78].

The most important limitation of Merkle trees, however, is in the resource requirements for maintaining the tree. As an example, consider a binary tree using MD5 as the hash function with the data blocks representing a typical cache line of $64B$. Every tree level L (where $L = 0$ represents the root) will have $N_L = 2^L$ nodes. Each node is a full MD5 hash occupying $H = 16B$ of data, and thus each level occupies $S_L = N_L \cdot H$. Summing over all the levels gives us:

$$S = \sum_{L=0}^{T} 2^L \cdot H = H \cdot \sum_{L=0}^{T} 2^L = H \cdot (2^{T+1} - 1)$$

The total number of levels given D, the total number of pages, which we assume is a power of two, is:

$$T_D = \lceil log_2 D \rceil = log_2 D$$

Which means the total size needed for the tree is:

$$S_D = H \cdot \left(2^{\log_2 D + 1} - 1\right) = H \cdot (2 \cdot D - 1)$$

In the context of the example given above ($64B$ data block, $16B$ hash size) the hash tree would require around 50% of the protected memory size. This excessive amount of space can be reduced at the cost of memory bandwidth consumption and/or processing speed. The arity of the tree (i.e., the number of child nodes hashed to obtain a parent node) can be increased, leading to fewer tree levels at the cost of requiring more nodes to be brought from memory when verifying each parent. The size of the data blocks hashed at leaf level can also be increased, with the same trade-off – more data needs to be brought from memory for integrity verification.

4.2. Performance optimizations

The primitives and algorithms described in section 4.1**Error! Reference source not found.** can be d eployed as described to satisfy the goals described in section 3.2. Our stated security goals are not, however, the only measures that determine adoption. As explained in section 2.5, performance degradation due to cryptographic memory protection is a major barrier to deployment, particularly in IoT devices. This degradation – as experienced by users or software engineers working with the platform – would primarily appear as longer execution times for the same tasks. The degradation has two main causes:

a) **Processing done on the critical path of memory transaction**: every memory transaction is now bound to extra processing before it becomes usable by the processor, from encryption/decryption of system data to integrity verification of all relevant tree nodes. This penalty can be reduced using the improvements described in sections 4.2.1, 4.2.3, and 4.2.4.
b) **Extra traffic necessary for the cryptographic protection**: performing the cryptographic processing described previously requires the type of metadata mentioned in section 4.1.1.1. This metadata usually resides in main memory and thus needs to be fetched along with the system data requested by the processor, which in turn leads to bandwidth amplification. The performance degradation caused by this amplification depends on the state of the system as a whole – in an unloaded system, the effect can be minimal, however as the memory bus becomes saturated the degradation increases exponentially (see chapter 7 for more detailed figures).

This is the reason why most of the research on cryptographic memory protection (this thesis included) has performance improvements as the main goal, all the while attempting to preserve the same level of security as previous iterations or designs.

The following subsections describe in some detail the various performance improvements that have been proposed, prototyped, and deployed in the last two decades. Their relevance to the baseline system, on which our new techniques described in chapter 5 have been implemented, is also discussed.

4.2.1. Specialized cryptographic primitives

While memory protection can be ensured by using a cipher such as AES in some mode of operation that has all the required characteristics and using a separate MAC algorithm to generate an integrity tag, this approach would be suboptimal given the available wealth of primitives optimised for specific use-cases. Lots of research has focused on producing strong primitives, schemes, and protocols targeted at important application fields.

Tweakable Block Ciphers (TBC) [69] are a good example of this research, incorporating features of modes of operation straight into block ciphers. Instead of creating a "higher-order" block cipher by using a mode that incorporates metadata, TBCs include a third input, a tweak, which parameterizes the cipher. The cipher is fully secure even in the case when the tweak is under the control of the attacker – the only purpose of the extra input is to provide variability to the ciphertext even when the plaintext is the same. The tweak can thus be used to include information about the memory location being encrypted. This covers the first of the two desired features presented in the previous section.

QARMA [70] is a TBC designed to cover the second feature – to offer a fast, highly efficient hardware implementation. Intended for memory encryption and other in-processor use cases, it has so far been deployed in pointer authentication within Arm cores [71]. QARMA draws inspiration from previous TBCs designed for low-power environments, PRINCE [72] and Midori [73], and comes in two block sizes – 64 and 128 bits, with the key twice the size of the block. The 64 bit variant is intended for applications with lower security requirements while the larger, 128 bit variant is meant to be sufficiently secure for most use cases including in the context of quantum computing. Figure 16 shows a comparison between QARMA and AES in terms of processing latency and implementation area. The *Minimum Area* and *Minimum Delay* columns show the figures for implementations of QARMA that had those respective goals. QARMA_r represents an implementation of QARMA with $2r + 2$ rounds. As a comparison, AES-128 consists of 10 rounds, while AES-256 consists of 14. σ_x represents QARMA using one of 3 S-Boxes (an internal component that has an important role in the security and performance of the cipher), with σ_2 being the slowest and most secure and σ_0 the fastest but least secure.

Targeting	*Minimum Area*			*Minimum Delay*		
	Delay	Area		Delay	Area	
Cipher	*ns*	μm^2	GE	*ns*	μm^2	GE
QARMA$_9$-128-σ_0	7.15	2321.4	42423	3.79	3844.6	70260
QARMA$_{10}$-128-σ_0	7.95	2528.2	46203	4.16	4225.8	77226
QARMA$_{11}$-128-σ_0	8.71	2732.2	49931	4.53	4607.0	84192
QARMA$_9$-128-σ_1	7.43	2461.0	44975	4.03	4430.6	80969
QARMA$_{10}$-128-σ_1	8.15	2707.2	49475	4.40	4866.0	88926
QARMA$_{11}$-128-σ_1	8.88	2947.9	53872	4.80	5301.4	96883
QARMA$_9$-128-σ_2	7.38	2552.4	46645	4.18	4632.5	84658
QARMA$_{10}$-128-σ_2	8.12	2804.2	51246	4.60	5087.7	92977
QARMA$_{11}$-128-σ_2	8.84	3027.4	55325	4.99	5520.3	100883
AES-128, pipelined	15.67	3894.1	71164	*Note: The latency of one*		
AES-256, pipelined	21.99	5533.7	101128	*full AES round is* 1.58 *ns*		

Figure 16 - Comparison of speed and size of QARMA and AES hardware implementations [70]

Combining confidentiality and integrity in one algorithm or scheme leads to authenticated encryption (AE). While MACs and symmetric encryption algorithms can be combined in a mix and match fashion to obtain AE, the consensus has gravitated towards strictly defined schemes. These schemes avoid subtle security issues that could be introduced by unfortunate combinations of algorithms. TLS 1.3, for example, restricts these schemes to a handful of Authenticated Encryption with Additional Data (AEAD) algorithms. Such schemes also allow the algorithm to be computed in one go, lowering the overall latency of the computation. Galois/Counter Mode (GCM) [75] is an example of such a scheme defined on top of an 128 bit block cipher. It combines CTR mode for confidentiality and a Galois Message Authentication Code (GMAC) for integrity. The GMAC is obtained by XOR'ing a hash of the ciphertext with an encryption pad.

Similarly, the specialization process that has occurred in the symmetric encryption ecosystem with the appearance of block ciphers and modes of operation designed with specific applications in mind, specialized AEAD schemes have been developed. One relevant scheme in our case is Qameleon [76] which builds on TBCs – QARMA in particular – and is designed with low-latency in mind, specifically targeting memory encryption as an application. By relying on tweakable ciphers, Qameleon has an easier task of incorporating metadata that prevents two categories of attacks: splicing attacks and replay attacks.

Our prototype does not truly perform the cryptographic algorithms on the data stored in memory. Instead, advanced hardware modelling of the primitives mentioned above is used to simulate the timing delays expected. More details about this approach can be found in chapter 6.

4.2.2. Metadata caching

The issuing of repeated requests to memory for metadata used to verify memory blocks is the root of both performance degradation causes highlighted above. An obvious solution is thus to attempt to reduce the number of such requests by storing, on chip, (meta)data which might be relevant for future system requests.

In the case of integrity tags, system cache lines are generally large enough to store more than one such tag (AES-GCM [75], for example, produces a 128-bit integrity tag, while system cache lines for high-end IoT devices easily reach 512 bits). Assuming some degree of spatial locality, each cache line could thus store the integrity tags covering multiple system requests.

Caching is even more important for improving the efficiency of the integrity tree. Given how integrity verification using such a tree works, the closer a tree node is to the root, the larger the memory area it "covers" – i.e., the more likely it is that a random system data block will depend on it for integrity verification. This is compounded by the fact that, once a cached tree node is reached during verification, the walk up the tree can be stopped, as we can rely on this node to be valid.

There are a couple of ways in which caching can be handled:

- The system cache can be used to handle the integrity metadata in the same way as system data. This approach saves space on-chip by sharing an existing cache; this approach could, however, reduce the overall performance of the processor if the cache pollution introduced by this sharing generates more traffic than it saves (via extra system cache misses). According to [79], in such a configuration up to 50% of the system cache could end up occupied by integrity tree nodes when using a regular Merkle tree. Caching of Merkle tree nodes was first presented in [80] (reporting a performance overhead of 25% with caching, vs up to 1000% without) and subsequently used in other designs [81] [77] [79] [78].
- The MPE can be enhanced with its own cache(s) intended for integrity metadata. This approach was initially used for the counters needed for freshness during encryption and integrity verification of system data [81] [78]. It was first proposed [82] following experiments in which a system with a larger system cache (used for storing counters as well) is shown to fare worse than one with a dedicated counter cache. Subsequently, [68] pivoted the design towards one in which the integrity tree nodes were themselves composed of counters and reserved a dedicated cache for them. This model was also followed by other designs [83] [84].

[85] looks into the effect of caching the various types of metadata (counters used for encryption/integrity verification of system data, system data MACs, and integrity tree nodes) using different caching strategies, and concludes that sharing the system cache for all metadata types is the most effective for their system configuration.

Our baseline configuration makes heavy use of caching, with dedicated caches for system data MACs and for integrity tree nodes. More details can be found in chapter 6.

4.2.3. Parallelization

Attempting to tackle the first performance degradation cause listed above, one possible approach is to allow some of the steps involved in the confidentiality and integrity process to be executed in parallel. A standard Merkle tree update is an example of an algorithm which must be performed serially – starting from the leaf-level, child nodes must be hashed to produce a parent node, which must in turn be hashed to produce its own parent and so on. Parallelization can be tackled separately for confidentiality and for integrity.

4.2.3.1. For confidentiality algorithms

For confidentiality, an improvement can be made by allowing the actual decryption to overlap with the memory access for fetching the system data. When direct encryption is in use (as described in section X), the data block must first be fetched before it is sent through a cryptographic accelerator for decryption. An alternative is the use of counter mode encryption.

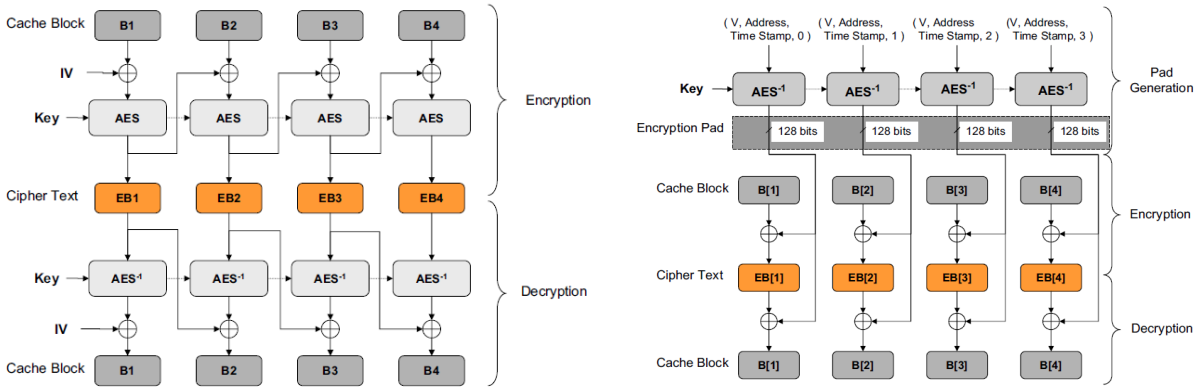

Figure 17 - Comparison in process between direct encryption and pad-based encryption [77]

Counter mode works by directly encrypting blocks of metadata and then XOR'ing the output (called encryption pad) with the plaintext to obtain the ciphertext. The metadata can be formed of a unique, random nonce with an appended counter, as shown in the figure, or by simply using a strictly monotonic counter. In either case, the nonce/counter must be available when decrypting, so *must* be stored in memory along with the block it helped encrypt. Another advantage of modes based on encryption pads, such as CTR, is shown in Figure 17 – the pad generation step (shown in the figure on the right) can be completely decoupled from the actual encryption or decryption step. The critical path is thus shortened since only an XOR *must* be performed on the critical path. The tag computation can happen in parallel with the memory access (assuming all the data necessary for it is available or retrieved in time). Figure 18 below shows how this parallelism can affect the latency of an encryption, compared to direct decryption.

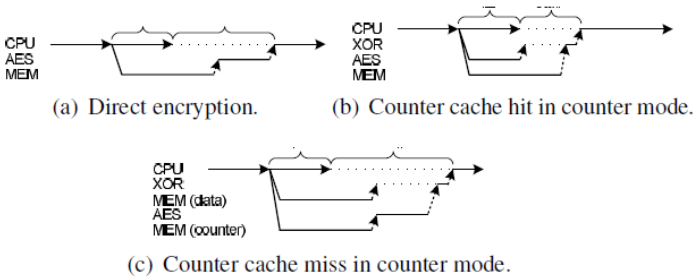

Figure 18 - Difference in latency between direct and pad-based encryption (i.e., counter mode) [81]

This technique was first proposed in [82] and used in most following designs [81] [79] [78] [68] [83] [84], as well as in our baseline prototype.

4.2.3.2. For integrity algorithms

In the case of integrity, parallelization can be devised, independently, at two different levels: when calculating or verifying integrity tags, and when performing a walk up the integrity tree. While

techniques exist to parallelize the former (by, for example, using similar mechanisms to those in section 4.2.3.1 to generate MACs by encrypting hashes with encryption pads), most research has focused on tree designs that allow parallelized updates (Merkle tree verifications can be performed in parallel).

The Parallelizable Authentication Tree (PAT) [86] was the first to propose a solution that allows parallel updates to the tree. Instead of only storing a hash/MAC for each integrity node, PAT trees are made from nodes formed of a nonce and a MAC. However, instead of deriving the MACs from the whole value of the child node like in a Merkle tree, the MAC is instead calculated over the nonces of child nodes, and the nonce of the current node. This allows, all nonces to be updated simultaneously and then the MACs recalculated.

The Tamper-Evident Counter Tree (TEC-Tree) [87] uses a different approach to achieve the same goal. It relies on the diffusion property inherent in direct encryption schemes to verify the integrity of tree nodes. The nodes are composed of counters, with each counter serving as a nonce in the child node. Both parent and child can be decrypted and checked that the counter equivalence holds. Like in the case of PAT trees, all the counters/nonces across the tree can be updated and then encrypted in parallel.

Newer tree designs, especially those based on counter nodes such as the one in Intel's MEE [68], take an approach similar to both the PAT and TEC-Tree, by using a counter in the parent node to generate a MAC (with freshness) in the child node [83] [84]. Since each MAC only depends on data found in cleartext in the two nodes involved, the computations can be performed in parallel across the whole tree after the counters are updated. Our approach also falls under this category of counter trees.

4.2.4. Asynchronous execution

The algorithms discussed so far rely on all the cryptographic checks and computations to finish before passing the data on to either the CPU or memory. Some of these checks can, however, be removed from the critical path and performed asynchronously, in the background. This usually comes at the cost of some security-guarantees – a laxer enforcement of integrity verification (even if the checks are only delayed a short amount of time) can lead to sensitive data being exfiltrated. Some designs allow speculative operation – AEGIS [77], for example, allows (in some circumstances) data to be released before being verified. Others, like Intel's SGX [68], do not allow speculation.

One proposed approach that stretches this trade-off towards maximum performance is Log Hash [77]. This allows the CPU to maintain a log of its memory footprint that allows verification to be triggered either from software or regularly using a timer. The application is thus free to enforce integrity verification on whatever granularity it requires without taking a large performance hit for the rest of its segments.

PoisonIvy [88] presents a design meant to secure asynchronous verification by tracking the use of unverified data through the processor, preventing any data based on speculation (including memory addresses) from leaving the processor.

An orthogonal technique relies on prediction and precomputation: [89] presents an approach for reducing latency by speculating the value of counters used for decryption/verification, while [90] relies on frequently-used values in memory to precompute speculated ciphertexts and/or MACs, which can then be compared with those retrieved from memory to avoid extra latency.

Our prototyping offers results both with and without asynchronous verification of incoming data, albeit without the type of enhancements provided by PoisonIvy.

4.2.5. Integrity tree size and arity

The shape, size, and inner working of the integrity tree is of particular importance, as it can have an oversized effect on the performance of the system. Caching and parallelization, as described in the previous subsections, can give a performance boost, however the design of the tree itself is an equally large component in the overall picture. Most importantly, the arity of the tree is a prime cause of bandwidth amplification: larger arity leads to shorter trees where each node covers a larger span of the protected memory.

The following subsections explore some of the noteworthy designs that have had a significant impact on reducing the size of the integrity tree.

4.2.5.1. Bonsai Merkle Trees

While the previous paragraphs discuss integrity protection of data alone, we must also consider the metadata needed for confidentiality purposes. As mentioned in section 4.1.1.1, encryption of each data block relies on a unique counter which must also be stored in memory in a separate metadata region. Correct decryption of data, however, depends on the integrity of these counters, and so the Merkle tree protection described in the previous section must stretch over the metadata region as well. The advent of authenticated encryption modes (discussed in section 4.1.2) brought forward an avenue for improvements. Since the counters used for encryption could also be used to generate integrity tags for the data blocks, replay protection could be ensured through these tags if the counters are themselves protected. The Merkle tree could thus be shifted to cover the counters only, forming a much smaller tree, as shown in Figure 19. This structure is known as a Bonsai Merkle Tree (BMT) [91].

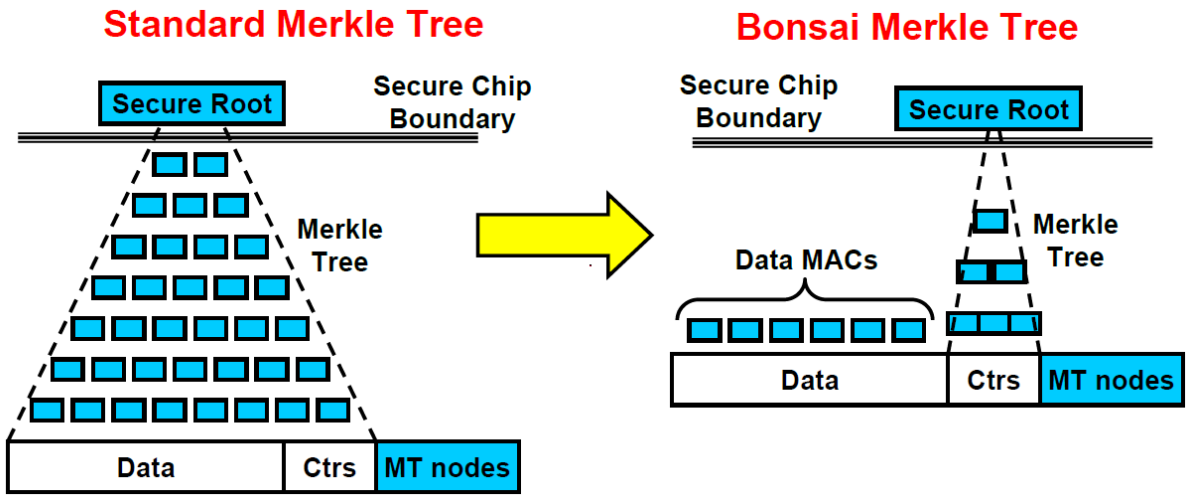

Figure 19 - Comparison between a standard Merkle tree and a Bonsai Merkle Tree [91]

4.2.5.2. Counter trees

Yet another evolution, following the BMT, is to replace the hash-based integrity verification from Merkle trees with a MAC-based solution, similar to the change that made way for BMTs. Instead of treating the two regions of memory (i.e., the data and counter regions) differently, counters can instead be grouped into data blocks and authenticated using another counter residing in the parent

node in the tree. A simplified example can be seen in Figure 20 below, coming from the Intel implementation of SGX. Here, the 4 counters found in the leaf level node $L00$, denoted $n00$ through to $n03$, are each assigned to a data *unit*, $U0$ through to $U3$. The counters are used when encrypting the data, and when computing the integrity tags associated with it. At the same time, to guarantee its integrity, $L00$ itself is treated as a data unit with its integrity tag ($Tag00$) saved along with the counters. This tag is computed using a counter stored in the parent node of $L00$, namely in $L10$. In this simplified example, only two tree levels are required, with $L10$ essentially residing on-chip and thus needing no protection.

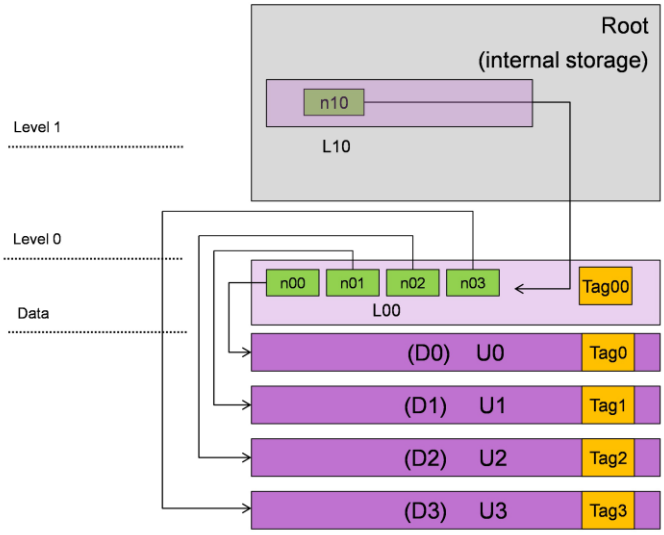

Figure 20 – Simplified implementation of a counter tree in Intel SGX [68]

Counter trees come with a few advantages over hash (or MAC) trees:

- Updates can be performed in a parallel – see section **Error! Reference source not found.**.
- Reduced data bandwidth overhead required for verification and updates – in a hash tree, to verify any integrity node, its parent and all its peers (all other child nodes belonging to the same parent) must be present to be hashed together. In the case of counter trees only the parent node is necessary, for obtaining the associated counter.
- Increased flexibility - by transforming the opaque, binary blob that represented an integrity node in a hash tree into a structured set of counters along with an integrity tag, more freedom in terms of data structure design is acquired.

4.2.5.3. Other counter tree designs

The flexibility acquired in the move from hash trees to counter trees can express itself in a plethora of ways, from the design of the data structure within the integrity node, to the exact mechanism for computing the integrity tag of said node. The design which serves as the base for the work presented in this thesis is, by comparison, simple: replacing the simple list of counters inside the IN with a set of split counters (Figure 21).

Despite the complexity of the diagram, the logic behind split counters is rather straightforward. Split counter mode encryption was first proposed in [81], where some consideration was given to the complications they create. Instead of storing each counter in its full representation as in the SGX example above, the counters are instead assembled by concatenating two separate counters – a

large, singular per IN counter (henceforth called Major counter), and a small counter, specific to each child node (henceforth called Minor counter). Figure 21 shows how one such IN would be used and authenticated. Its major counter, C, is shown next to 64 minor counters, labelled c_0 through to c_{63}. In order to obtain the counter used for encrypting and/or verifying the integrity of the first child node, for example, C is concatenated with c_0 to obtain the full counter C_0. C_0 is then used in the same way as in the SGX example above.

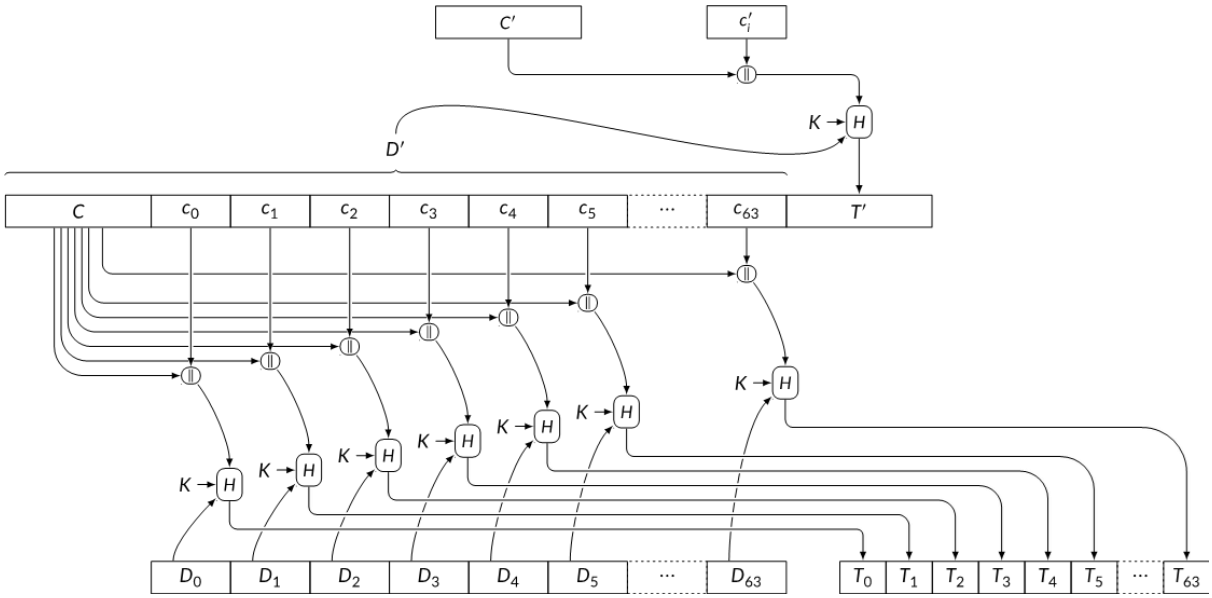

Figure 21 - Design of an integrity node in a split counter tree [133]

While split counter trees reduce the overall space required for the tree (in the example above, 64 64-bit counters can be compressed into 512 bits if the major counter is 56 bits long, the integrity tag is 64 bits long, and each minor is 6 bits long), it comes at a cost in the complexity of the algorithms required. This is due to the need to keep all counters strictly monotonic, and thus a requirement that an overflow in a minor counter (i.e., reaching its maximum value and needing to be increased again) must result in an increase of the major counter. However, since all other minor counters (which have not overflown) must remain consistent with the encryption and/or integrity verification of their child node, all child nodes must be brought in from memory to be re-encrypted and to have their integrity tag recalculated using the newly incremented major counter. This whole process describing the update performed for a peer counter of an overflowing minor will henceforth be called Read-Modify-Write (RMW). The example above would require 63 RMWs for each minor overflow – a huge amplification of memory traffic, especially for cases when the memory bandwidth is particularly under stress.

VAULT [83] is another proposed design using split counter trees, with the twist being that the arity of tree nodes varies along the height of the tree to reduce overflows in nodes found higher-up.

Our baseline prototype uses a split counter integrity tree with constant arity. More details can be found in chapter 6.

4.2.6. Dynamic tree designs

Arity is not the only property of an integrity tree that can be adjusted. The VAULT design mentioned above attempts to boost performance by statically adjusting the tree design so that not all INs share

the same configuration. Other designs exist which attempt to take this approach a step further, shaping the tree to the running workload.

Among the first papers to cover unbalanced integrity trees, [92] introduced a design in which a Merkle tree was shaped to shorten the path for specific memory areas that are expected to be used more frequently. The path to tree root for the rest of memory is, however, increased, while the overall size of the tree is kept equal to the balanced version. [93] furthers this design idea by proposing a method to statically adapt an (unbalanced) TEC-Tree [87] shape to the expected workload, with embedded devices as the main target. [94] and then [95] define this space further by proposing a tree design (again based on TEC-Tree, but applicable to other types of trees) which can perform dynamic skewing, along with a framework which combines this new design with architectural enhancements which can aid the system performance. [96] is a similar design which allows dynamic splitting and skewing of the integrity tree at runtime.

A separate approach is taken in [84], which achieves a larger arity than the counter trees discussed in section 4.2.5.3 by dynamically adjusting the size of the counters within each IN. Counters are compressed when unused/unmodified and dynamically sized after their first use.

All these designs achieve better performance figures at the cost of higher complexity cost. We have chosen to, instead, focus on the simpler static, balanced counter trees.

5. Contributions

Having reviewed the state and evolution of cryptographic memory protection, this chapter stands as an extension to section 4.2. We go into the theoretical details of the performance contributions we have brought in this thesis relative to the baseline prototype taken from [4]. We first go into the details of new techniques contributed by this thesis, followed by an overview of techniques which have been proposed previously and which we have implemented and tested. The chapter begins, however, with a closer look at the performance issues that we have tried to address, at how they arise, and at how the relationships between them influence our solutions.

5.1. Addressed issues

Establishing the distinct issues we have tried to address is an important step before defining our solutions. Section 4.2 defined two main areas of concern that affect the performance of cryptographic memory protection. However, the relationship between some of the mechanisms proposed in the subsections of 4.2 and those areas of concern is not always straightforward. The following subsections attempt to pinpoint the exact issues we have tried to overcome.

5.1.1. Bandwidth consumption

The bandwidth amplification described above happens on both reads and writes: whenever a new memory block must be brought from or written to DRAM, up to $N + 1$ blocks of metadata (where N is the height of the integrity tree) must also be fetched and/or written back. On top of this, the use of split-counter trees leads to RMW operations (see section 4.2.5.3), in which a single write to memory can give rise to tens or hundreds of extra requests.

This extra bandwidth consumption can be (somewhat) kept in check using on-chip caches for the metadata required by the cryptographic operations. For example, the baseline system that this thesis has been built on allows the caching of integrity nodes and of system data MACs separately [4]. A limitation of this approach is the decreasing amount of temporal locality in memory requests experienced as one goes further down the memory hierarchy – each level of caching necessarily removes some of this locality, since the caches closer to the processor already contain the data.

Another approach is to simply reduce the amount of data or metadata that must be brought in from memory. Split counters are an example of a mechanism that follows this goal, reducing not just the metadata footprint of each system data block, but also the number of tree levels that might be required, through the increase in tree arity. Nonetheless, they come with the drawback of increased bandwidth pressure whenever minor counters overflow.

Previous experiments have shown that, in the context of a saturated memory bus, the performance overhead of cryptographic memory protection is substantial. In conjunction with the pressure created by minor overflows, this limits our ability to increase the arity of the integrity tree.

Similarly, even though at most one extra memory request is necessary for retrieving the MAC of a system cache line, their size (and therefore density per metadata cache line), correlated with the reduced temporal locality mentioned above, makes MACs a particularly thorny problem on a saturated memory bus.

5.1.2. Memory consumption

While not as pressing as the performance issues presented previously, the requirement to allocate a proportion of the memory for metadata storage places a burden, particularly on devices with limited resources such as IoT devices. Reducing this requirement can prove beneficial in terms of bandwidth usage, as described above, but also by increasing the memory available to the running workloads. At the same time, a low memory footprint can open the doors for other types of optimizations, as shall be described in the following section.

5.2. Newly proposed mechanisms

The techniques and mechanisms described in this section are novel approaches meant to solve one or both issues highlighted above. They have been translated into prototypes, with the results and analysis available later, in chapter 7.

5.2.1. Multi-level split counter trees

As discussed earlier, split counters (implemented as seen in Figure 21) provide significant improvements to integrity tree arity and to the size of temporal metadata required for system data. While both improvements amount to a reduction in bandwidth consumption in the general case, RMW operations can put significant pressure on the memory bus when the system is under load.

Following the progression from monolithic counters to split counters, an alternative way to structure the data in an integrity node is to split the counters once more to produce multi-level counters (see Figure 22).

Instead of concatenating a minor counter to the end of a major counter shared by all the minors within the IN, this approach creates groups of counters, each group associated with a middle counter. Each counter is then formed by concatenating the major (common to all counters in an integrity node), a middle (common to a group of counters in the integrity node), and a minor (specific to that counter). Whenever a minor overflows, its associated middle counter is incremented, and all its peer minors must be reset using RMW operations like those for simple split counters. The grouping of minors underneath a middle therefore acts to limit the number of RMW operations per minor overflow. Whenever a middle counter overflows, all middle and minor counters must be reset and the major counter incremented, leading to the same number of RMW operations as for simple split counters.

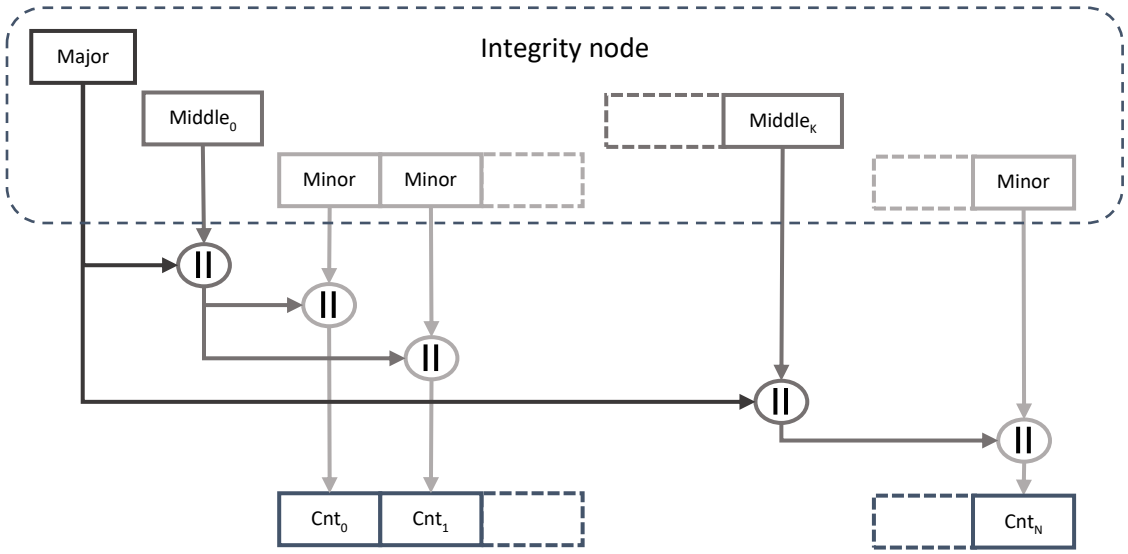

Figure 22 - Counter construction in a multi-level split counter integrity node

Given that the space used for middle counters could, instead, be used to extend the minor counters and thus delay overflows. This means a balance must be struck between the reduction allowed by grouping the minors, and the increase caused by shorter minor counters.

Structuring INs in this way opens the way for another improvement: increasing the arity beyond what was sustainable with simple split counters. A 512 bit IN can, for example, be divided into a 64-bit major counter, 16 4-bit middle counters, and 128 3-bit minor counters (see results in section 7.3.1).

5.2.2. Dedicated MPE on-chip memory

The previous sections have all looked at one avenue to performance improvement – reduction of metadata bandwidth usage on the memory bus. Such reductions can only go so far until the security of the system is affected. Stepping back from this approach, the main issue appears due to usage of an expensive commodity (performance-wise) – the bandwidth of the memory bus – for metadata required for a security feature. A more sustainable solution is to relocate the traffic to a separate bus, keeping the main memory bus for regular system traffic. The optimal case is for this bus and the dedicated memory it connects to be within the same integrated circuit as the processor and the MPE (see Figure 23, and contrast with Figure 10). Coupled with some of the techniques highlighted previously, the dedicated memory can hold a significant amount of metadata.

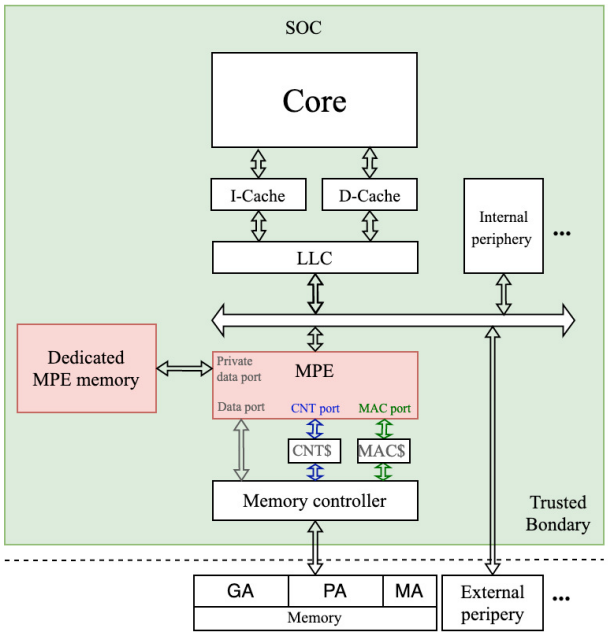

Figure 23 - Architecture with MPE with dedicated memory

The nature of the extra memory can provide benefits on its own, given that all data stored within the chip can be considered secure and does not need extra protection.

If integrity nodes are stored on chip, they require no extra integrity protection, and thus the MAC bits can be re-allocated for other purposes (e.g., adding middle counters, as described in section 5.2.1). Through the same reasoning, the integrity tree is no longer necessary in securing the leaf counters, and only the leaf-level INs need to be kept as temporal metadata for system data. This arrangement has a beneficial effect in multiple ways:

c) Less memory is required to store all temporal (and anti-replay) metadata.
d) Less time is taken to verify the integrity of a data granule, as only its MAC (or hash, see below) needs to be checked.
e) Less bandwidth is taken to verify the integrity of a data granule, since there are no tree-level INs to be pulled from memory.
f) No cache space will be taken by tree-level INs, and thus the cache will be able to cover a larger proportion of memory through leaf-level INs.

If integrity tags are stored on chip, they no longer need to be encrypted and can, instead, be stored as plain hashes of the data granules. The processing time for integrity verification is therefore reduced.

All these differences which cascade from properties of on- and off-chip memory to the functioning of the security mechanisms lead more concretely to differences in the state machines of the MPE. These differences as described in the subsections of 7.2.1 as appropriate.

5.3. Prototyped existing optimizations

Unlike the approaches described in the previous section, the techniques described below have been previously proposed as improvements, without an analysis of its effect in a split-counter integrity tree. This thesis bridges that gap in available results by integrating these techniques in our baseline prototype and benchmarking them.

5.3.1. Split counter rebasing

Counter rebasing has been proposed in [84] as one of the improvements to their MorphCounter integrity node algorithm. This new IN configuration was compared with a simple split counter configuration as described in section 4.2.5.3. The rebasing technique can, however, also be applied to split counters with a minor modification of the counter construction mechanism. Instead of concatenating minor to major, rebasing requires that the two are combined in a linear way, for example by adding them together: $C_x = C_{major} + C_{minor,x}$.

A high-level summary for counter rebasing is that, instead of performing RMW operations when a minor counter overflow occurs, a rebase of the minors is first attempted. If all counters have been incremented at least once since the last overflow, they can all be rebased on top of the minimum counter value. For example, in the case where the combination mechanism between major and minor is simple addition (as described above), the minimum minor counter value is subtracted from all minors and added to the major counter. All the counters are, thus, still the same value, however the minor overflow is averted. An example can be seen in Figure 24, and a more thorough description can be seen in [84].

Prototyping results were presented in [84] for counter rebasing, but only when applied to a MorphCounter IN configuration. In this thesis we look at rebasing in the context of simple split counters instead.

	Major Counter	Minor Counter (3-bit)				Effective Value (*major + minor*)			
Overflowing Minor Counter	100	5	6	8	7	105	106	108	107
After Overflow (Existing Design)	108	0	0	0	0	108	108	108	108
Avoiding Overflow With Rebasing	105	0	1	3	2	105	106	108	107

Figure 24 - Results of handling an overflow with and without counter rebasing [84]

5.3.2. MPE with system cache integration

The large contributions to bandwidth amplification are not limited to metadata being transacted between the MPE and memory. RMW operations at the leaf level of the integrity tree, for example, involve requesting tens of system data cache lines from memory to be re-encrypted and have integrity tags regenerated. The requested data can, however, be found on-chip, in the caches that serve the processor. An improvement avenue is thus to insert a link between the MPE and the system cache, allowing the exchange of data and/or other signals.

5.3.2.1. For RMW operations

The proposed integration between the MPE and the system cache would allow us to probe, for each leaf-level RMW, whether the system data due to be re-encrypted is in the system cache. If the data is present and dirty, we can be sure that a future write will be required to that address. A re-encryption is thus not necessary. Alternatively, we could set the dirty flag on the cache line to ensure it is always written back.

Complications could arise if, for example, other cores could end up reading the old data from memory – its integrity verification would fail. We assume the only way to access the data protected by the MPE is through the system cache, and thus through this protection engine.

This was first proposed in [81].

5.3.2.2. For multi-cache-line MACs

Another generic optimisation technique that has not been discussed so far is the expansion of system data granule size over which MACs are computed. Thus, instead of having a MAC for each protected cache line, a number of cache lines (e.g., 2, 4, 8, …) are grouped together to compute one single MAC. The process relies on combining partial hashes computed over each cache line and then encrypting the result. [83] and [4] go into more detail about this process of computing integrity tags. This comes as a trade-off between overall performance (since verifying each MAC ends up requiring all its MAC-peers, on top of the baseline metadata) and memory usage.

In the same vein as the optimization for RMWs described earlier, the data necessary to verify MACs can be retrieved from the system cache, if available (and not dirty). Whether the optimization can or needs to be applied to MAC updates (i.e., when one of the cache lines needs to be modified) depends on the mechanism used when combining the partial hashes and producing the MAC: with the right mechanisms (e.g., when the partial hashes are XORed together to form the granule hash), the partial hash for the modified cache-line can be updated independently, using only its old and new hashes, and the old MAC value.

6. Prototype assessment framework

6.1. Systems prototyping

Developing new features for integration into processors and systems-on-chip is an expensive and error-prone endeavour with multiple layers of risk. On one hand, deploying new features improves the existing designs and allows for more complex and useful use-cases. On the other hand, any mistakes or issues identified only after production can turn out to be extremely expensive. The problems could also turn out to be impossible to fix, since the use of updates – so widespread in the software industry – is not an option when it comes to hardware (though software updates might help mitigate the issue at the cost of performance and/or security, for example in [97]).

A solution for this risk lies in prototyping – the creation of an approximate implementation of the desired features that can be integrated, tested, and benchmarked on various metrics. Prototyping is usually done in several increasingly accurate models. For example, once the abstract functionality is agreed upon, a new CPU feature might begin life as a software simulation that gives a rough estimate of the performance overhead introduced by the feature. Trade-offs and design ideas can quickly be assessed, and the promising designs can move on to a low-fidelity hardware prototype intended to verify the complexity of a hardware implementation. The process can continue, going back-and-forth between different prototyping levels if issues are discovered further in the development process.

Another major use of prototyping is as a measure or differentiator of ideas. Proving the validity of a new design by integrating it into an existing system is more convincing. A similar approach can be taken to compare competing ideas, especially in cases where modularity of the prototyping framework means that each implementation can easily be slotted into a wide array of system architectures. The performance of new ideas can then be assessed and compared on devices with various levels of capabilities, from small microcontrollers to large, multi-core systems.

Prototypes can thus be considered important tools in the Research and Development toolbox, critical for lending credibility to new ideas. Many of the research papers cited in the previous sections ([71] [83] [81]) bring forward not just ideas but also (results from) prototypes of those ideas. A similar approach is taken in this work – new ideas are assessed based on their benchmarking results.

6.1.1. Software vs hardware prototyping

As briefly described in the previous section, system prototypes can come in different formats and are intended for different goals. One divide exists in the medium of the prototype, namely software prototypes versus hardware prototypes.

Software prototypes treat the system as an application to be run, which simulates in software a sufficiently realistic approximation of the target hardware platform. The development and testing of such models are typically cheaper and easier thanks to the lightweight toolchains available for software development and given that tests can be carried out on any computing platform. Another benefit comes as the ability to instrument the model to obtain significant amounts of fine-grained details and statistics about its functioning. A major downside is represented by the speed of execution – due to the large amounts of computation needed to simulate each CPU instruction,

software models can perform hundreds or thousands of times slower than a comparable hardware prototype.

Hardware prototypes, on the other hand, are intended to replicate the functionality in hardware, at some level of abstraction. The new functionality is introduced into an existing system and the result can be benchmarked. Testing usually involves a type of programmable integrated circuit which can be configured to function as the synthesized prototype. Such prototypes can offer a better feel for the complexity of a production-level solution and a finer estimate for performance cost of the implementation but are more difficult to test and more complex to work with.

This thesis focuses on software prototyping for a few reasons:

- The more rapid development lifecycle allows for more ideas to be investigated.
- It allows for easier mix-and-matching when assembling the system, which means a wider range of possible system designs can be tested.
- A software prototype already existed within Arm for cryptographic memory protection, developed for the exact purpose of investigating different approaches to the problems described in the previous chapters. This model is built in the gem5 framework which will be discussed in the next section.

6.1.2. Prototyping in gem5

As mentioned previously, the prototypes used for investigating the ideas proposed in this thesis are built in the gem5 simulator, “a modular platform for computer-system architecture research” [98]. Gem5 allows engineers to build software versions of hardware components typically included in computer systems. The framework also helps abstract away the interfaces between components. The components can thus be combined programmatically and configured at runtime. A few of the parameters that can be configured are:

- Architecture of the system processor
- Number of CPU cores
- Number and sizes of processor caches
- Algorithm choice for controlling the internal behaviour of different components (e.g., the replacement policy of caches)

Gem5 prototypes take as inputs executables intended for the configured processors and execute them, yielding detailed information and statistics about the system as it runs, and about the entire run as a whole. After each simulation, statistics are collected from each component involved in the system, outputting all of them into a file. Since the number and type of such statistics is under the control of the developer, any required metrics can be obtained with high precision. Some examples of metrics that can be obtained:

- Number of CPU instructions executed
- Amount of data that passed through a specific memory bus
- Number of times a specific event took place within a component (e.g., number of times a request sent to a cache resulted in a hit)

6.2. System setup

The MPE that forms the object of study of this thesis exists as an intermediary block between the CPU and its primary memory (Figure 10). The system around the MPE therefore needs to be defined,

as it is against the baseline performance of this unprotected system that the impact of the MPE will be assessed. All the components are implemented as Gem5 timing components, configured, assembled, and run using a script.

- **Processor**: the whole system revolves around the use of a single-core Arm v8-A CPU. To keep consistent with previous simulations performed using this MPE model, the Arm A57 core was chosen [99]. The A57 is an out-of-order superscalar processor, and it usually features as the big core in a big.LITTLE configuration. It features two levels of caches, with the last level cache (LLC) being common to all cores (in multi-core configurations). While not particularly common in real-world processors, the A57 closely resembles several other Arm cores which have seen wider adoption, particularly among mid- and high-end IoT devices. For example, the RaspberryPi 4 [6] features a quad-core A72 which is the successor of A57 and presents many of the same features.
- **Caches**: the A57 core comes with two levels of caches – the first level is split into two, a 32kB L1D cache (for data) and a 48kB L1I cache (for instructions), while the second level can be anywhere between 0.5 and 2 MiB. For this thesis, the size is kept at 1MiB throughout. Using clustering in the CHI interconnect, the effective system cache line size is of 128 bytes.
- **Main memory**: the external memory being simulated is a single channel DDR4-2400 x64 device in an 8x8 configuration. This gives us 16GiB to work with, partitioned into a protected memory range and an unprotected range. Workload traffic is always directed towards the protected range.

A more comprehensive description of the system can be seen in Table 2.1 of [4].

Once assembled, the system can be provided a specially-crafter binary for it to run to completion. During the execution of this binary, each component in the system registers and measures several metrics relevant for the simulation, which are collated at the end of the test and output in a text file. These metrics are then processed to obtain aggregate data that covers the results of multiple runs or binaries. In the case of this thesis, data processing is done using the Matplotlib [100] and Numpy [101] Python libraries.

6.2.1. Prototype architecture

The cryptographic memory protection support described in chapter 4 has been implemented in gem5 as a standalone component whose placement within the system aligns with that of Figure 10. From a high-level perspective, the MPE accepts requests for data located at specific memory addresses and returns that data to the component that requested it, while also enforcing the security services required of it. If the MPE detects an integrity violation affecting either an IN or a data block, a signal is issued to the processor to notify it of the incident. Typically, the requests come from the last level cache over a bus and are returned over the same bus. The internal architecture of the MPE can be seen in Figure 25 below.

The MPE can be seen as composed of the following components (a more detailed description of their operation can be seen in the following subsections):

- **Crypto logic**: handles the main processing loop of the MPE, keeping track of all the requests coming from the processor and their current state. It interacts with the other MPE components to obtain all the relevant data, performing the cryptographic operations to decrypt system data and verify its integrity, or to encrypt and generate integrity tags.

- **Counter logic**: handles all the logic pertaining to the integrity tree. It serves as a black box for the crypto logic, returning specific leaf counter values which will then be used for seeding the cryptographic operations. In the background it performs all the steps necessary to verify or update the state of the tree, using its associated cryptographic accelerators and counter cache.
- **MAC cache**: handles the integrity tags of data blocks, including partial hashes, if necessary.

The MPE is implemented as a Gem5 timing component, meaning that its main purpose is to model the delays incurred by its operation. This approach also has other implications on the implementation of the components mentioned above, which will be mentioned in the following subsections.

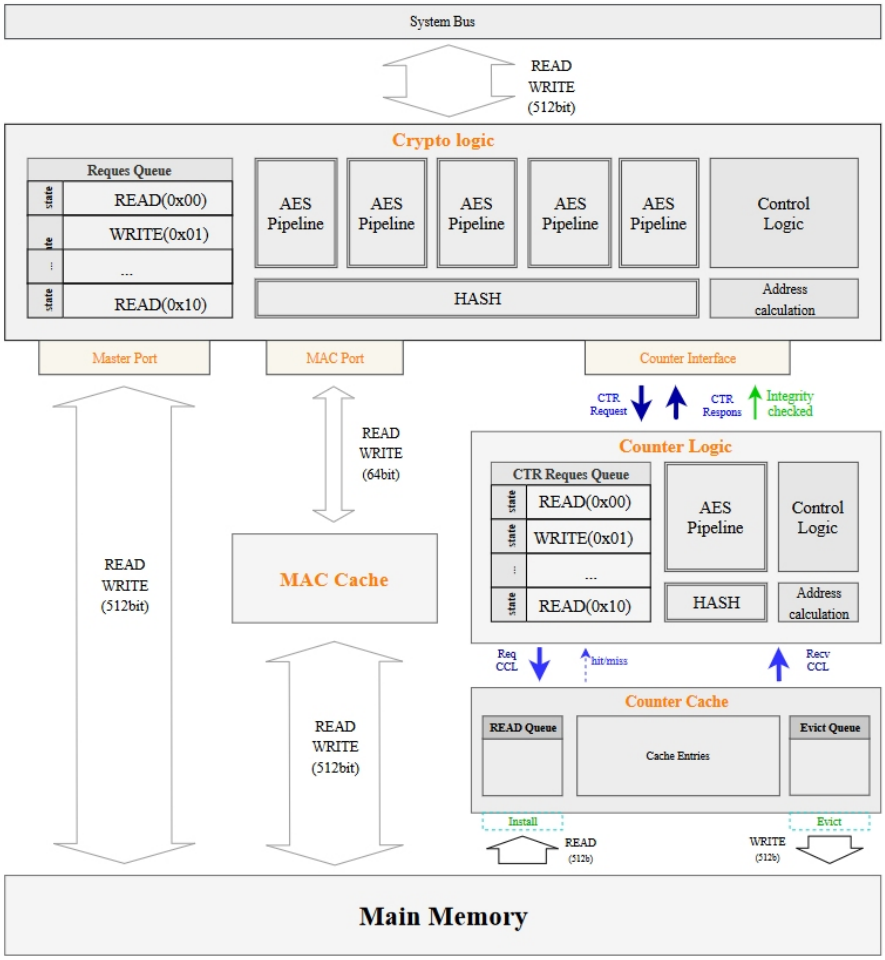

Figure 25 - High-level architecture of the Memory Protection Engine [4]

Each of the components relies on finite-state machines (FSM) to maintain the state of the requests or operations currently in-flight. While a full description of these state machines is not given here (it can be found in [4]), a brief overview is provided below.

6.2.1.1. Crypto logic

The Crypto logic block represents the main driver of the MPE. It maintains a combined queue of read and write requests for data blocks formed of one cache-line each. Each such request is carried through an FSM while all the necessary data is pulled from main memory, and all processing is carried out. The two types of requests have different processing needs and thus different FSMs.

For read requests, the data block must be read from memory along with its MAC and with the counter used in during encryption and MAC generation. When some of this data becomes available and depending on the current state of the request, certain bits of processing can begin. For example, when the counter is retrieved from the Counter logic block, the encryption pad can be generated. The cryptographic operation is, however, never executed – only its duration is simulated. Due to the nature of the MPE (a Gem5 timing model), it is irrelevant whether the encryption or integrity verification is indeed performed, if the delay attributed to it is realistic. This simplifies the model and allows the effect of different ciphers/hardware implementations to be easily compared, simply by swapping the expected delay values for their duration.

Another important factor in the case of read requests is the option of **asynchronous integrity verification**. Typically, when a maximum level of security is desired for the device, data cannot be returned from the MPE until its MAC has been verified. (This also applies, as shall be described later, to the Counter logic.) When the security requirements can be relaxed to gain a performance boost, the read data can be released as soon as it is decrypted, leaving any integrity violations to be reported after being identified. This obviously comes at the risk of an attacker injecting tampered values into the system. As shall be seen in section ?, the improvement in performance can be significant.

For write requests only the counter must be obtained from the crypto logic, as the new data comes with the request. Unlike read requests, however, the counter obtained must be a fresh value, having been incremented by the counter logic. In the case where the integrity nodes contain split counters, as described in section 4.2.5.3, incrementing a counter could result in an overflow, and thus a need for RMW operations. In such a case, the counter logic also provides a list of addresses that must go through an RMW. To finish the initial write request, the data can then be encrypted, and a MAC can be generated.

For each required RMW, the crypto logic issues a special type of read request. After following the steps necessary for a normal read, the unencrypted data is re-encrypted, and a new integrity tag is generated.

6.2.1.2. Counter Logic

As described previously, the Counter logic block implements the algorithms powering the integrity tree. Much like the crypto logic block, the counter logic revolves around a queue of requests – either read requests (i.e., return the current counter value for a given address), or write requests (i.e., obtain the current counter value for a given address, increment it and return it, resolving any overflow). These requests follow FSMs similar to those in the crypto logic, as integrity verification is also needed for integrity nodes.

For a read request, as detailed in section 4.2.5.2, the leaf level integrity node which contains the requested counter must be verified. This involves a walk up the integrity tree, verifying the IN from each tree level using its parent node, until we either find one of the nodes in the counter cache (in which case we can trust its integrity, as we have verified it previously), or we reach the root of the tree (which is stored on chip).

During the walk up the tree, each node fetched from memory and verified is then installed in the cache. Thus, another node must be evicted from the cache to make room for the new one. If the evicted node was dirty (i.e., at least one of its counters has been modified since the last read from memory), then the integrity tag of the node must be updated before writing the IN back to memory. This kicks off a write request, such that the parent of this node will be brought in from memory if it is

not present in the cache – the corresponding counter must then be extracted from it, incremented, and used to generate the integrity tag of the evicted IN. The parent must in turn be verified, which could in turn lead to more evictions. The counter in the parent could also overflow, leading to yet more requests. The counter logic is implemented to reduce the number of such avalanches of request.

A write request follows a similar flow, with the extra step of incrementing the counter before sending it to the crypto logic.

In both cases, whenever an overflow occurs, a list is assembled of counters that must be reset, and thus of child nodes that need to undergo an RMW. If the overflow occurs at leaf level, the list is sent back to the crypto logic, whereas if it happens within the tree, the list is used to simply issue new requests within the counter logic block.

6.2.1.3. MAC cache

The MAC cache is far simpler than the crypto logic or the counter logic blocks. Its only purpose is to store integrity tags for system memory blocks, in unencrypted form. When receiving a request from crypto logic, it returns can return a notification that the tag was found in the cache, in which case the data block hash can be directly compared to the cached hash.

One possible optimisation for the MPE is the use of integrity tags that protect data blocks which stretch over multiple cache lines. In such a case, the MAC is calculated based on the combined hashes of the cache lines forming the data block. The MAC cache then holds onto not just the overall integrity tag for each block, but also to the partial hashes, i.e., the hashes of the underlying cache lines. These values can be used when updating the MAC – if one of the underlying lines gets written back to memory, the integrity tag can be updated using its new hash and the “old” partial hashes from the other lines, instead of requiring those lines to be read from memory.

6.3. Benchmarking

Measuring performance of a computing platform raises a question about the measurement process – it is important not just what metrics we compare, but also what produced the numbers. The hardware on its own does not have meaningful performance figures associated with it, as the figures need to represent the execution of some software components. The task, thus, comes down to deciding which software components should be used to profile the hardware. Unfortunately, the answer to this depends on what exactly the user of the device will use it for, and since computing platforms are, normally, general-purpose, no unique use-case can be expected. More abstractly, this thesis is not aimed at a specific use case, even in the IoT field. A database server and an image processing service can be running in parallel on the same IoT Edge gateway, both with different memory access patterns. Our memory protection mechanism must thus be as performant under as many different types of software tasks as possible. Benchmark suites have been created with such profiling in mind, covering different types of software products that might be encountered on a computing device.

6.3.1. The SPEC benchmark suites

Some suits filling this gap have been created by the Standard Performance Evaluation Corporation (SPEC) to profile devices on compute-intensive tasks. The SPEC2006 [102] and SPEC2017 [103] suites include tens of real-world software products being used on standardized inputs – stretching from file

archiving tasks to graph path finding algorithms. The 2006 version represented the de-facto standard for generic single-threaded performance. The 2017 version was an update meant to cover not only different interests in terms of software use-cases, but also a change in the points of contention for hardware. While IoT-specific benchmarks (such as IoTBench [104] or EdgeBench [105]) exist, the SPEC suites were the workload choice for this thesis for a couple of reasons:

- **Availability of workloads**: as mentioned previously, prior to running any binaries using a Gem5 model, they must be processed to align with the execution model of the framework. Along with a baseline MPE implementation, Arm has also provided a library of pre-processed SPEC 2006 and 2017 benchmarks. A difficult problem to overcome in the context of running real-world benchmarks on Gem5 models is the amount of time taken for the benchmark to run. As was shown in [106], SPEC2006 benchmarks could take a month or more for one such run. A solution comes in the form of workload characterization. The original workload (say, one of the gcc compilation workloads that are part of SPEC2006) is analysed and "compressed" into a series of short code segments that best represent the overall characteristics of the workload [107]. Each segment is also associated with a weight value that quantifies how representative the segment is of the whole workload. An estimate for the overall result of the original workload is then obtained through weighted averaging of the results for the traces. The results for this thesis were obtained using a library of such code segments (henceforth called SimPoints, or traces) for the SPEC2006 and SPEC2017 suites.
- **Availability of previous results**: given that this thesis is building on previous work undergone within the Arm research group, using the same benchmarks allows better comparisons with results obtained for configurations that have already been profiled.

6.3.2. Realistic environment simulation

The performance characteristics obtained from workloads used as described above offer good estimates for a system where the benchmark is the only running process. This is, however, not a realistic model for the software environment in which such workloads would run. Even more modest devices can and do run multiple workloads simultaneously, including an operating system, other supporting service, as well as parallel workloads. A more realistic simulation environment is thus beneficial in understanding the effect of the MPE on performance.

A major shortfall of running independent workloads is the lack of stress on the memory bus. Most workloads do not claim enough memory bandwidth to test the effect of MPE-led memory amplification in a high-contention case. If, for example, minor overflows start occurring as described in section 4.2.5.3 when the memory bus is already saturated, massive delays could be incurred by the whole system. Such situations are, however, not properly represented by running single benchmarks. The solution is to also simulate memory traffic as coming from other software components, in parallel with the benchmark. The exact characteristics of this traffic are presented later, in chapter 7.

6.3.3. Metrics of interest

Benchmarking assumes the presence of specific metrics that show how the impact of the MPE varies with respect to the variables available in configuring the system, both hardware and software. In the case of this thesis, the two types of metrics are most relevant:

- **Processing speed**: measured as either the average number of processor cycles required to successfully execute each instruction in the workload (Cycles Per Instruction, CPI), or as the average number of instructions executed per cycle (Instructions Per Cycle, IPC). CPI will be used for most results in this thesis. These measurements provide a generic measure of the performance degradation brought by the MPE.
- **Memory usage**: measured either as memory amplification (how much the number of reads and writes from/to DRAM increase when an MPE is used), as memory bandwidth increase (average increase in read/write/total bandwidth during the workload run), or as increase in memory traffic exchanged with the DRAM. Memory processing data can shed light on the details behind a decrease in processing speed, revealing the cause and potential improvement areas.

6.3.4. Previous results

Since the work in this thesis builds upon previous published results, it is worth looking back at what research directions have been followed and what results they have brought up. Their broad investigation topics can be summed up as: performance impact of increasing protection level; generic optimisations; impact of split counter trees. Each of these topics is discussed more broadly below. For detailed insight into the results, refer to [4].

6.3.4.1. Performance impact of increasing protection level

Protection levels are characterized by the security services they provide. The levels are additive – each level gets more security properties assigned, on top of the ones provided by the previous levels. Despite not being relevant to this thesis, the options represented by the simpler protection levels is worth keeping in mind in terms of the trade-off between security properties and cost incurred by the user.

- **Level 1 (L1)**: memory confidentiality alone. At this level all data in memory is encrypted without any integrity protection. No freshness is included, so the same plaintext is always encrypted to the same ciphertext (if stored at the same memory address, given spatial metadata, see section 4.1.1.1). This is by far the simplest to implement and has an insignificant performance impact. However, it only defends against a passive attacker who can read from the memory bus/memory chip, without changing any values.
- **Level 2 (L2)**: adds integrity verification to L1. Each data block gets its own MAC that can help detect changes made by an attacker. The integrity tag includes no freshness either, so the same plaintext will always have the same MAC, if stored at the same address. This level also does not provide any protection against replay attacks (see section 4.1.2.1). Performance degradation, compared to L1, is noticeable.
- **Level 2+ (L2+)**: adds freshness to encryption and integrity checks. Along with the MAC, each data block has a counter associated which is used for freshness. Passive attackers cannot tell whether the same plaintext was stored at the same address anymore. Replay attacks are, however, still feasible. Performance degradation compared to L2 is, again, noticeable, particularly for memory intensive or sensitive workloads.
- **Level 3 (L3)**: adds a counter tree to prevent replay attacks (see section 4.1.3). Performance decays even more compared to L2+, as expected due to the extra traffic occurring. The difference is most pronounced when the memory bus is at or near saturation.

This thesis is only concerned with optimizations for L3 implementations, however whether L3 is the best option for a specific IoT device is debateable, based on the expected use case and threat model.

6.3.4.2. Generic optimisations

These optimisations apply to all MPE implementations that aim for level 3 protection. In a similar vein, some of the results presented in chapter 7 fall under the same categorization.

- **Asynchronous mode**: As was described in section 6.2.1.1, the integrity checks necessary to validate the integrity of a data block can be performed "in the background", after the data has been released to the system following decryption. This improves the performance of the system but reduces the security guarantees as malicious data could enter the processor and be used before an integrity check failure is flagged. On average, the CPI during asynchronous runs was roughly 10% lower than in synchronous runs.
- **MAC caching**: Given that MACs for data blocks are stored in a separate memory area, and since multiple MACs can be fit into each cache line, holding on to recently read blocks of MACs is a natural idea. This exploits spatial locality in memory accesses to improve performance. The performance improvements see diminishing returns as the cache size grows, following an exponential decay, with slow improvement past the 1-2kiB mark.
- **Counter caching**: Similarly, spatial locality strongly benefits the caching of integrity nodes. The benefit is even more pronounced in the case of split counters, since the number of counters per IN can be at least an order of magnitude larger than in monolithic counters, thus covering larger proportions of system memory with the same cache size. The performance improvement follows a similar pattern as for MAC caches, with a different inflection point around 32-64kiB.

6.3.4.3. Impact of split counter trees

Split counter trees (as described in section 4.2.5.3 and in [108]) were implemented and tested as a comparison point against an implementation modelling a monolithic counter approach. Multiple configurations pertaining to the arity of various sections of the tree were tested, with the most efficient serving as the baseline for the model used in this thesis (see chapter 7). This baseline configuration achieved as low as 4% geometric mean overhead over an unprotected system for the whole set of benchmarks – though part of the improvement came from all the details and optimizations discussed in the previous section.

7. Prototyping results

The previous chapters have focused on describing the issue we are trying to solve, its roots, the progression of mitigations available in literature, and an overview of our contributions. The crux of this thesis lies, however, in the simulation results obtained from the prototyping work described in chapter 6. Before diving into the actual figures, a discussion is warranted on how they should be interpreted, and what their applicability span is.

The results presented throughout the remainder of this chapter have both an absolute and a relative component. The performance degradation figures we obtain for our system (as an increase in CPI, for example) can be viewed as a rough estimate of how such a component would affect performance in a real device. Limitations or optimizations applied when implementing a production-quality MPE could skew the actual figure either way. But at the same time, and perhaps even more so, these results should be read comparatively – just how much we can improve performance of a given design through thought-out modifications to its operation. This view also comes with its own caveats in precision. With this in mind, a short list of possible problematic factors is discussed for each optimization.

As has already been mentioned, our improvements are implemented, tested, and compared to a baseline MPE design taken from the work done in [4] (see Table 2). This component is simulated in a system (described in chapter 6) which has very specific characteristics in terms of processing capability, memory characteristics and so on – for example, the test system uses a single Arm A57 core. Such an approach limits the applicability of (some of) the results. The relevance of the absolute values in our results is particularly affected, while their relative dimension is more likely to stay consistent across a larger share of device configurations. This is important given the heterogenous world of IoT platforms, even when limited to just its high-end tier.

Table 2 - Characteristics of the baseline MPE configuration

Property		Description
Metadata cache line size		64 bytes
Integrity verification synchronicity		Asynchronous (i.e., speculative release of system data)
MAC	Size	32-bit MAC per system data cache line
	Computation model	UHF hashes are computed over encrypted system data; the encryption includes no freshness as this is already included in the encryption
	Dedicated cache properties	2 KiB, LRU replacement
	Crypto primitive used	QARMA-128 is used to encrypt the UHFs
Integrity tree	Node structure	Split counter with 64 counters per node: 64 6-bit minor counters, one 64-bit major counter, one 64-bit IN MAC
	Integrity checking model	A UHF is calculated over the IN, which is then encrypted using the parent counter as a tweak
	Dedicated cache properties	64 KiB, LRU replacement
	Crypto primitives used	QARMA-128 used for encryption of system data (tweaked using leaf counters) and for IN MACs; UHF used for integrity checking of INs

While Table 2 defines the baseline model as making use of speculative release of data, in some cases results for a synchronous system (i.e., with no speculation) might also be of interest. This will be mentioned accordingly.

Figure 26 and Figure 27 below show a normalised view of the performance degradation of our system with the baseline MPE described above, relative to the same system without an MPE for protection. For normalized CPI figures, *lower is better*. The normalization step entails dividing the result of the system under investigation by the result of the reference system – in this case, our figures with the MPE enabled are divided by those without an MPE. Many of the following graphs are structured in this way, using an unprotected configuration as the reference, to highlight both the absolute and relative changes in performance, as stated earlier. The difference between the two figures lies in the stress level applied to the system. In Figure 26, the benchmark under investigation is the only thing putting pressure on the memory bus, while in Figure 27 we inject extra memory traffic in the system just before the system cache. The extra traffic consists of 8 GiB/s of mixed reads and writes (75% reads, 25% writes), performed in two sequential steps: during the first step the reads and writes are performed linearly through a 1 GiB memory block designated for this purpose, while during the second they are performed randomly throughout the same memory block. All charts representing a loaded system (i.e., a system with enough pressure on the memory bus to bring it to saturation) use extra memory traffic generated with these parameters.

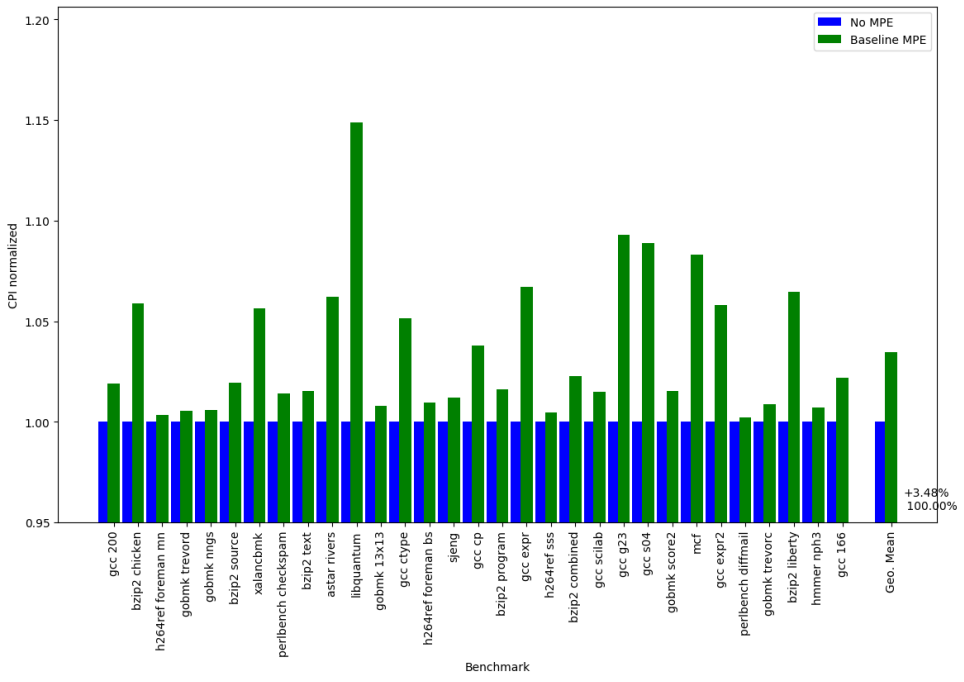

Figure 26 - Performance degradation due to the baseline MPE in an unloaded system, relative to an unprotected system

A different way to view and interpret the results presented throughout this chapter is by contrasting two types of figures – those which apply to one of the SPEC 2006 benchmarks, and those which are the result of aggregating the numbers obtained for all the benchmarks we run. A case can be made for the importance of both, with the former representing individual types of workloads that might run on (specialized) IoT platforms, while the latter represents the expected figures for general-purpose platforms. Thus, even if the aggregate figures see little-to-no improvement under a specific

optimization, a significant boost in performance for a benchmark which generally suffers from high overheads (e.g., libquantum) would be valuable.

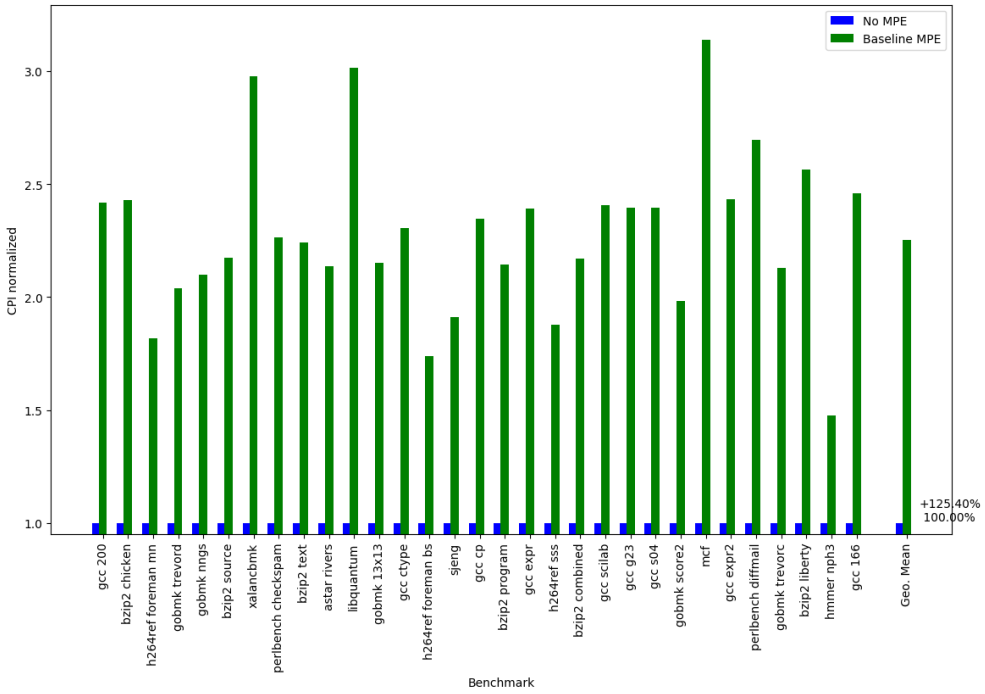

Figure 27 - Performance degradation due to the baseline MPE in a loaded system, relative to an unprotected system

7.1. Assessing counter tree optimizations

This section introduces results for our optimizations aimed at split counter trees. These optimizations mostly aim to reduce the overhead due to RMW operations. It is, therefore, useful to have a closer look at how much RMW operations figure into the performance degradation for our baseline system before we investigate our optimizations.

7.1.1. Measuring impact of RMWs

Many of the optimizations proposed and investigated in this section relate to reducing the number of Read-Modify-Write operations triggered by minor counter overflows in split counter trees. Given this goal, an important data point to establish is the upper bound on performance improvement. If we were to design a perfect system which would avoid all RMW operations, how much faster would that system be compared with one in which every RMW operation is executed? Figure 28 compares the performance of the baseline MPE with one which skips all RMW operations. Such an experiment is possible because the MPE does not actually perform the cryptographic operations on any data passing through, simulating instead the timing delays involved in the processing steps. This experiment covers both the unloaded and loaded system configurations in order to address the concern that, when the memory bus becomes a bottleneck, the RMWs lead to considerable performance degradation.

Our results show that, at least for our system and MPE configurations, the overhead of dealing with RMW operations is not significant even under increased memory bus stress. The overhead reaches between 2% and 6% (out of a total of between 100 and 200%) of the CPI of the loaded, unprotected system for some of the benchmarks – e.g., mcf, xalancbmk. Despite this observation, limiting the RMW overhead still bears some importance – the CPI and RMW counts shown above represent

aggregate values over multiple traces, while RMW operations cause performance degradations in bursts whenever minor overflows occur. These bursts still have the potential to cause significant

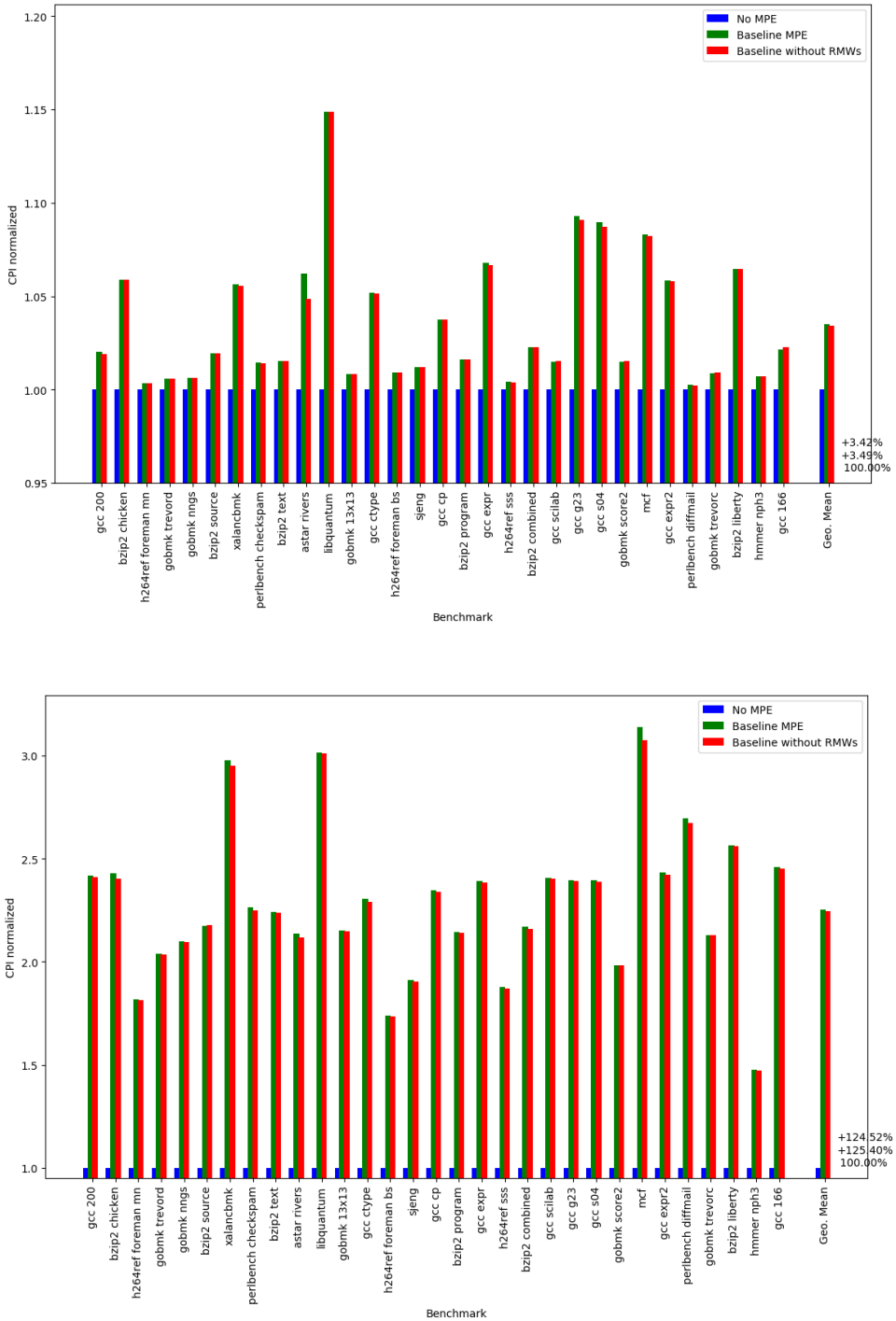

Figure 28 - Assessing the upper-bound on performance improvement through optimizations related to RMWs alone, by comparing our baseline to a system that does not perform any RMWs; the two figures represent the results for an unloaded system (above), and for a loaded one (below)

problems at threshold levels, where a handful of overflows can cause significant performance issues during a critical section of code. At the same time, while the impact of RMWs in our current system configuration is negligible, other optimizations (for example those discussed in section 7.2.1) could bring down the overheads to a level at which RMWs are a major performance polluter.

Another aspect worth considering is the distribution of minor overflows along the tree levels (level 0 corresponds to the leaves of the tree, containing counters which are used to directly protect system data, while level 3 is at the top of the tree, protected by the root counters stored within the MPE). Figure 29 charts this distribution for both a loaded and an unloaded system. Both the distribution and the absolute number of overflows differs widely between the two. For the unloaded case most overflows happen in the lower levels, driven by writes from the benchmarks. For the loaded system the vast majority of overflows occur in the middle of the tree, driven by constant switching between the areas accessed by the benchmarks and those accessed by the traffic generators leading to constant evictions of leaf-level INs from the counter cache. The loaded system also produces two orders of magnitude more overflows than the unloaded system. These distributions have a substantial impact on the effectiveness of the optimizations discussed in the following sections. For example, the system cache integration covered in 7.1.3 only affects leaf-level overflows and is thus unlikely to make a difference when only a small proportion of overflows occur in level 0.

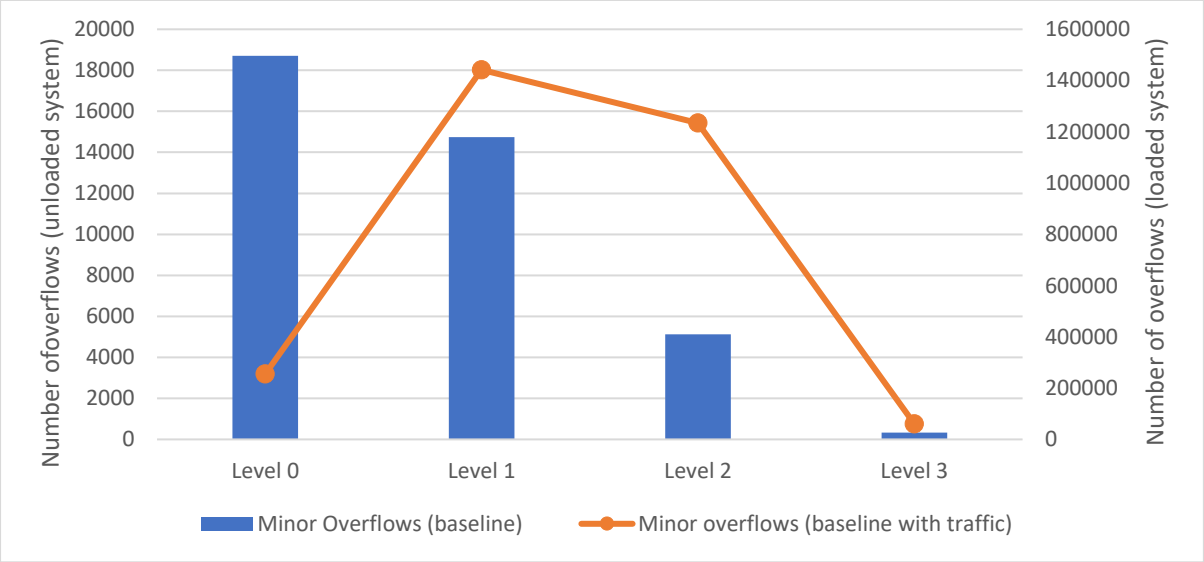

Figure 29 - Distribution of minor counter overflows across the levels of the integrity tree, for both loaded and unloaded systems; level 0 (i.e., leaf level) is used to protect system data; level 3 is the highest off-chip level, protected directly by the on-chip root

Given the limited relevance of RMW operations on CPI in unloaded systems, the focus in this section will mostly fall on comparing the number of RMW operations performed.

7.1.2. Multi-level split counters

Multi-level split counters represent a generalization of what we will henceforth call simple split counters – a way to organize the bits of an integrity node to form a hierarchy of counters. One step beyond simple split counters is our technique, described in section 5.2.1, where the counters are composed of a major counter (shared by all counters in the IN), a middle counter (shared by a group of counters), and a minor counter (specific to one counter). This new setup attempts to solve two problems, both related to the number of RMW operations necessary:

- **Lowering the average memory request burst size on minor counter overflow**: the extra layer of counters acts as a buffer of sorts, containing the impact of minor counter overflows to a subset of the counters in the IN. Thus, if a IN containing 64 counters has 8 middle counters, only 7 child nodes must go through a RMW when a minor overflow occurs. However, given that middle counters are also relatively small, the likelihood of them

overflowing is significant in the long run. When this happens, a full set of RMW operations must be performed (63 in the case of the example above), as all middle counters, along with all their associated minors need to be reset as well. These middle overflows are expected to occur much less frequently. *This is the main benefit of multi-level counters*.

- **Lowering the number of RMWs**: following the logic outlined above, the total number of RMWs could be reduced. This benefit is less clear-cut and more dependent on the exact memory access patterns of the applications involved, given that middle counters use up bits which could, instead, be distributed to the minor counters to prevent them from overflowing in the first place. For example, if 1 bit is taken from each of 64 counters and arranged into 8 middle counters, we can expect minors to overflow at least twice as often. Each minor overflow would then lead to 7 RMWs instead of 64, and for every 256 minor overflows (within a group) the middle counter will overflow, leading to 63 RMWs. The exact counts, however, depend on which counters end up being incremented more frequently, which in turn depends on the access patterns of the software running on the device.

Given the contrasting views presented by these two points, it is worth assessing both the performance impact of this IN design, and its effect on the number of occurring RMWs. Table 3 gives a comparison between the IN structure in the baseline MPE, and the triple counter design.

Table 3 - Comparison of the characteristics of multi-level split counters with the simple split counter used within the baseline MPE

	Baseline MPE	**Multi-level counter MPE**
Number of counters per integrity node	64	64
Number of counter levels in an integrity node	2	3
Sizes of counters in each level	1 x 64-bit major counter 64 x 6-bit minor counters	1 x 64-bit major counter 16 x 4-bit middle counters 64 x 5-bit minor counters
RMW burst sizes	63 RMWs for each 64 updates to one minor counter	3 RMWs for each 32 updates to one minor counter 63 RMWs for each 16 updates to a middle counter

Figure 30 shows the performance figures of our multi-level counters compared with the baseline, both in term of normalized CPI and in number of RMWs. The charts comparing the number of RMWs are plotted on a logarithmic scale to account for the large discrepancies between benchmarks.

The CPI charts show little effect in terms of actual performance improvement or degradation, even on a granularity of one benchmark. The number of RMWs shows more variation: some benchmarks see massive increases compared to the baseline MPE configuration (e.g., $\sim 100\%$ increase in gcc ctype). On the other end of the spectrum, the benchmarks affected by most RMWs in the baseline configuration see a large improvement when using a triple counter tree (e.g., $\sim 66\%$ fewer RMWs in astar rivers). The performance figures for a system with extra traffic are similar, with the overall CPI discrepancy at only 0.01% of the unprotected system CPI.

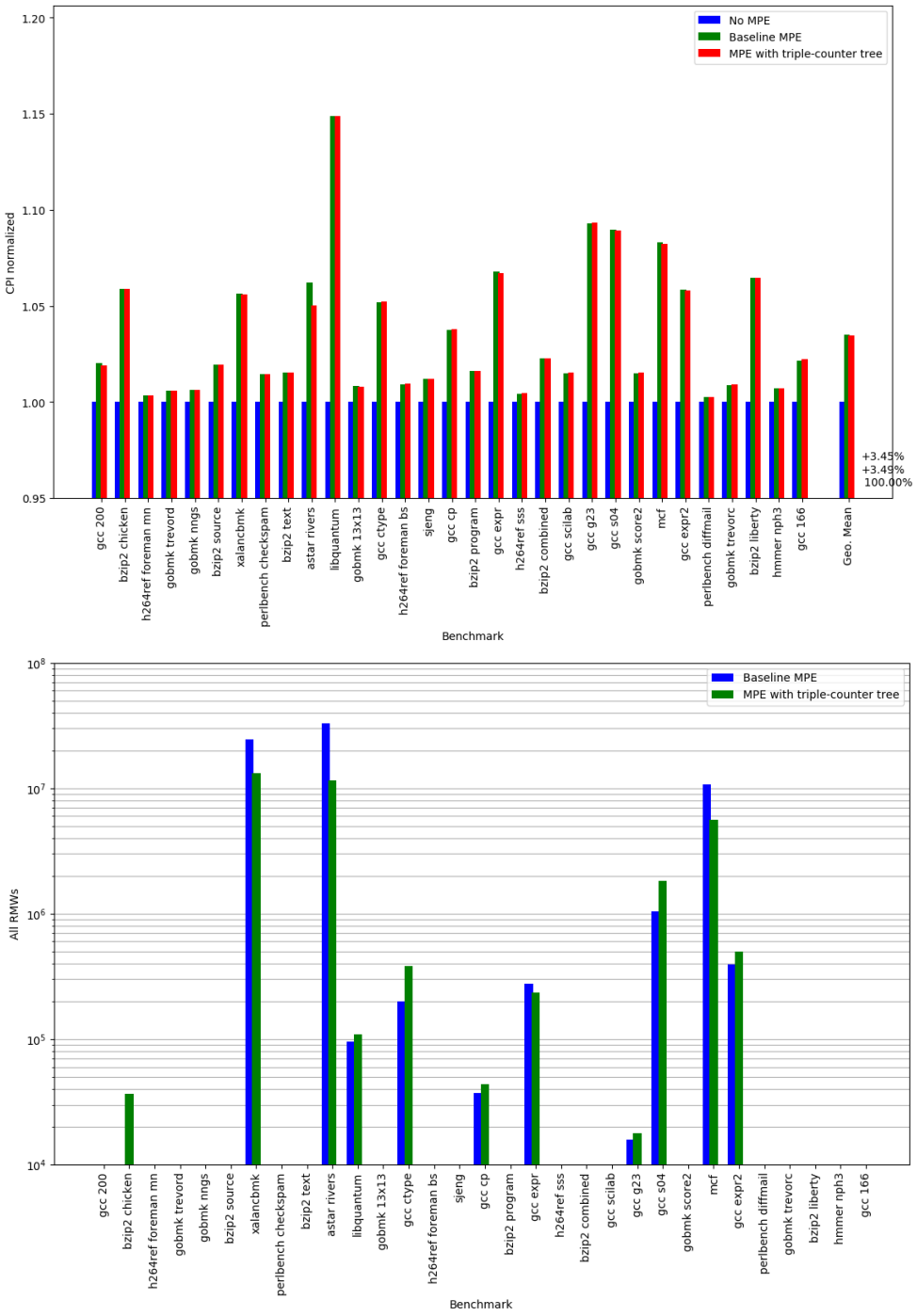

Figure 30 - Comparison of the performance characteristics of the multi-level split counter tree, relative to the performance of the baseline integrity tree; above - normalized CPI; below - RMW count (on logarithmic scale)

These results show that triple counters, on their own, bring no clear performance advantage. It is only in combination with other optimizations, such as those in section 7.2.1, that this new design proves advantageous.

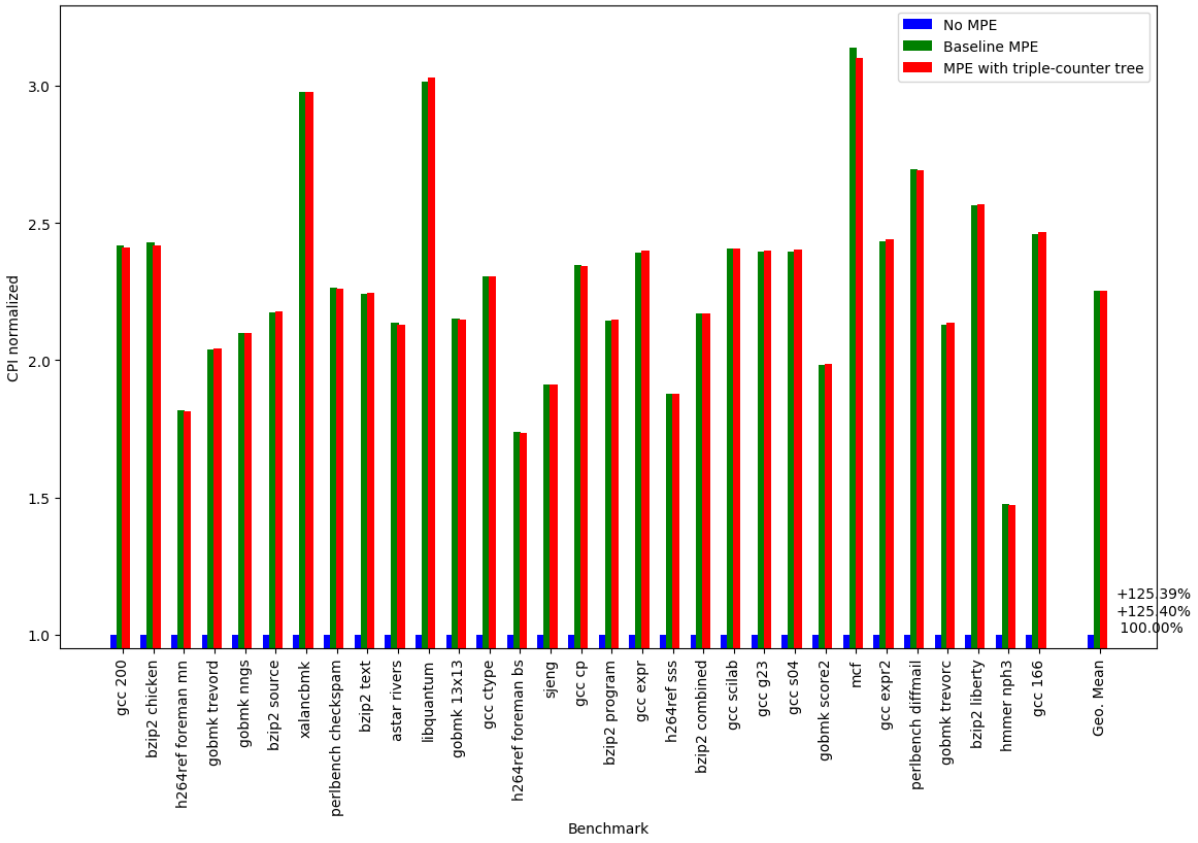

Figure 31 - Performance comparison under stress between MPE with multi-level counter tree and baseline MPE

7.1.3. System cache integration for RMW operations

One optimization aimed at reducing RMWs at leaf-level is the integration with the system cache presented in section 5.3.2.1. In theory, by checking the system cache for the system data cache lines which need re-encryption, this approach could reduce the number of such blocks by almost 50%, according to [81]. The limitation to leaf-level RMWs comes from the fact that neither integrity tree nodes, nor system data MACs are cached in the system cache – instead, we provide dedicated caches attached to the MPE.

Figure 32 shows comparisons between the number of leaf-level RMWs executed under the baseline MPE and under an MPE making use of system cache integration. The diagrams compare the results for both loaded and unloaded systems to compare the effects of the two access distributions. When only the benchmarks are executing, their execution alone determines the contents of the system cache, and therefore provide a better approximation of the effect of the optimization per application type. The traffic generators introduce a degree of randomness and bring the results closer to those on a fully functional system. The charts show a mixed bag of results – for most benchmarks in the system under stress the system cache integration had no discernible effect. On the other hand, for a handful of the benchmarks (most notably those based on gcc) this optimization lowers the number of RMWs by around 50% in both the loaded and unloaded experiments, and up to 75% for gcc expr

in the unloaded system. This optimization can thus be regarded as having limited functionality, mainly useful for specific types of software applications.

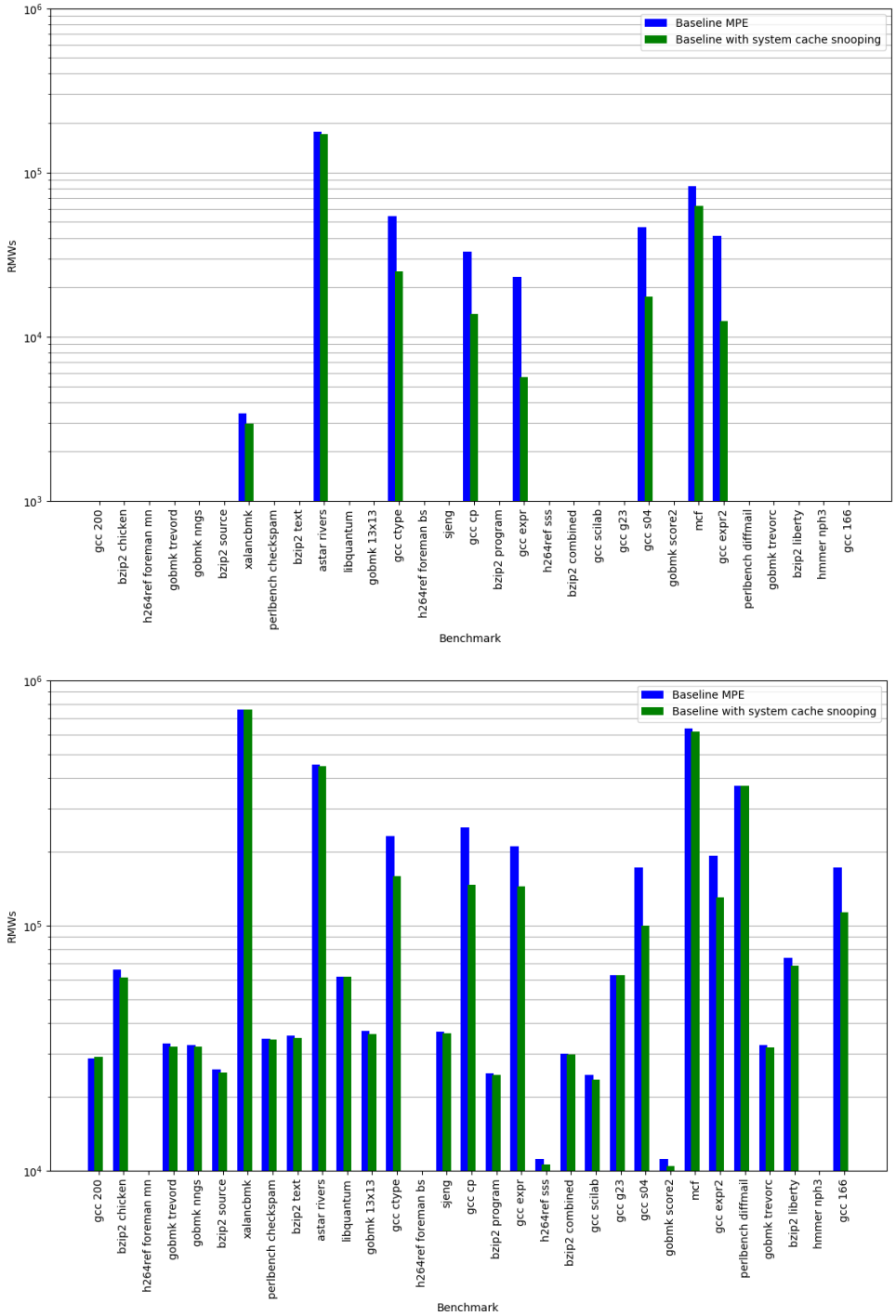

Figure 32 - Comparison in number of RMW operations performed between baseline MPE and an MPE with an integration into the system cache; above - results for unloaded system; below - results for loaded system (both have logarithmic scale)

7.1.4. Counter rebasing

The goal of counter rebasing, as presented in section 5.3.1, is to limit the frequency of counter overflows by attempting to shift down the values of all minor counters in an integrity node when such a shift is possible. Unlike the snooping optimization presented in the previous section, counter rebasing can be applied across the whole height of the integrity tree. Figure 33 shows the effect of counter rebasing both in the entire tree of a loaded baseline system, and separately in the leaf counters. The size of the performance improvement at the whole-tree level is significant for some benchmarks, but never overwhelming – the RMW count dropped by as much as 25%, most notably for the gcc benchmarks again.

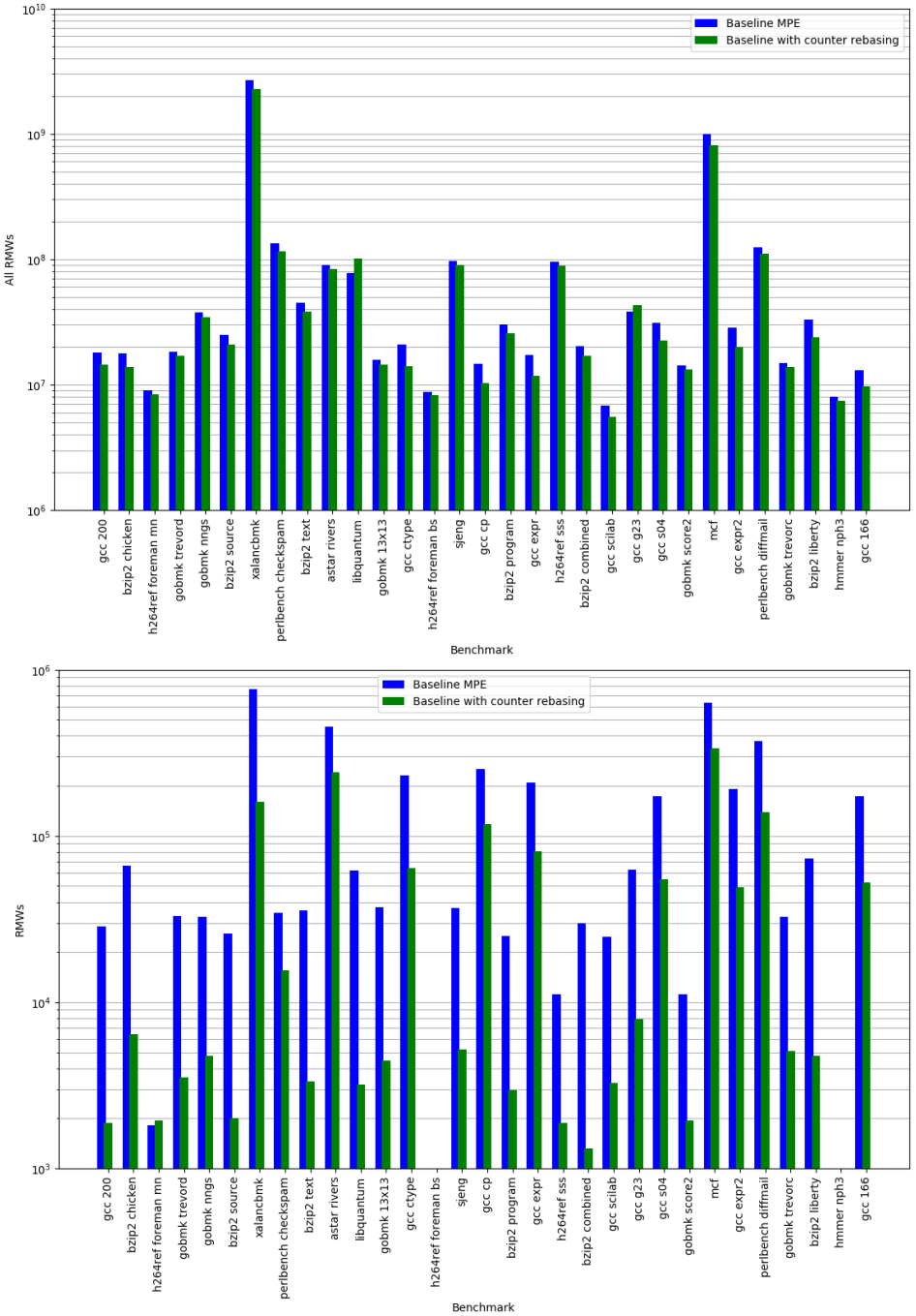

Figure 33 - Comparison in number of RMWs between a baseline system and one with counter rebasing; above - RMWs for the whole integrity tree; below - leaf-level RMWs only (both in logarithmic scale)

The improvement at leaf-level is, however, more spectacular. The RMW count drops by at least 50% in all benchmarks, and by as much as 95% in the case of libquantum.

This discrepancy between the improvement at whole-tree level and leaf-level is most likely due to the memory access patterns involved in the system under stress: counter rebasing relies on all the counters in each integrity node to have been incremented at least once when a minor overflow is due to occur. Given that nodes in each successive level up the integrity tree covers a wider and wider area of memory, this condition can only be met if the benchmarks and/or the extra traffic generated also sweep over more and more memory. Since the traffic generator acts on a block of 1 GiB of memory, and most benchmarks are fairly limited in how much memory they require, many of the overflows occurring in higher levels of the tree are unlikely to benefit from rebasing. This is a limitation of trees with a large arity – the counters within each level end up covering parts of memory with significantly different access frequencies yet must be kept in sync with each other via the common major counter. Figure 34 shows the breakdown of rebases among the tree levels and the relative decrease in number of minor overflows because of the rebases.

Figure 34 - Breakdown of minor overflows and counter rebases across the levels forming the integrity tree, for the loaded system; level 0 (i.e., leaf level) is used to protect system data; level 3 is the highest off-chip level, protected directly by the on-chip root

The relative number of rebases at leaf level compared to higher in the tree confirms our reasoning. Indeed, this is also in line with results from [4], where the values of minors in leaf-level integrity nodes were found to be more uniform than those in tree-level nodes. Unfortunately, as discussed earlier, the high amount of switching between areas being accessed leads to most overflows to be in the middle levels of the tree, making rebasing less effective overall. However, some optimizations presented in following sections can improve this effectiveness (see 7.3.2).

It is interesting to remark the large number of rebases at leaf-level, compared with the overall decrease in minor overflows. Indeed, the number of successful rebase operations is more than twice the number of previous minor overflows. This is possible because rebasing does not necessarily bring all minor counters back to 0, and so the number of minor counter updates required to trigger another rebase or overflow is lower (possibly as low as 1) than when an overflow is processed without a rebase.

7.2. Assessing generic optimizations

Section 7.1 has looked in detail at the results obtained for optimizations that target the functioning of the integrity tree. We now move on to more generic techniques, which could prove beneficial regardless of the integrity tree.

7.2.1. Dedicated on-chip memory for MPE

Primary memory can come in many sizes, technologies, and types of packages. Memory technology taxonomy has already been briefly touched upon in section 3.4. The types of memories which we are attempting to secure are discrete DRAM memories, found as external components to the CPU IC. These are generally used for their density and size, providing the system with plenty of space for memory-intensive applications. However, as mentioned in 3.4, the technology exists to combine memory and processor in the same IC – not by sharing the same substrate as SRAM blocks adjacent to the CPU, but by stacking them as a 3D structure called Package on Package (PoP). Low Power DDR (LPDDR) memory is one type of memory which can be efficiently combined with a CPU in this way. HBM [60] is yet another option which offers higher performance within a similar, 3D packaging model. Though the result is more expensive than allocating a similar amount of memory off-chip, the types of attacks which fall under this thesis' remit are mitigated, as communication paths between processor and memory are inaccessible for probing and the memory component cannot be easily separated from the processor. Given the evolution of industrial semiconductor production, it can be expected that on-chip memory will become more commonplace in the future, with larger, slower banks of memory available off-chip.

We have investigated how on-chip memory could be used to improve the performance of securing external memory. While HBM technologies would be ideal performance-wise, their current use-case is generally within the datacentre space given the cost of the technology [109] [110]. Our best bet for IoT platforms is, thus, PoP LPDDR technology. This section looks at the performance improvement we can expect from such a configuration. An important topic when assessing these optimizations is the fine balance between the expected performance gain and the amount of on-chip memory we choose to relocate from the external DRAM. Given the high cost of such memory, our goal here is not only to improve performance, but to do so at the least possible opportunity cost to the rest of the system.

The characteristics of the LPDDR memory we have used in our simulation can be seen below in Table 4. This memory block has been allocated for the sole use of the MPE block. This could be a reasonable deployment pattern for select IoT devices, where on-chip memory is added solely for securing off-chip purposes. More realistically, on-chip memory would most likely be shared with other components, either software or hardware, so the numbers available in this thesis serve as an upper-bound for the performance improvement.

An important thing to note is that the security characteristics of the on-chip memory affect our mitigations in a deeper way than simply shifting the bandwidth from one memory bus to another. These changes are described in section 5.2.2.

Unlike the RMW-related optimizations from 7.1, we expect the performance impact of this on-chip memory to be significant, and thus comparison of normalized CPI should suffice as a metric.

Table 4 - Characteristics of on-chip memory used in our simulations

Memory technology	LPDDR5
Size	1 GiB
Maximum bandwidth	5.5 Gbps
Number of channels	1
Bus width	16 bits
Burst length	32

7.2.1.1. For leaf integrity nodes

The first target for relocation to on-chip memory is the whole integrity tree. In this configuration, the term “integrity tree” is a misnomer: because data found on chip can be considered secure, any counters found there do not need any extra protection by, for example, building a tree structure on top of them. All we need to store on-chip are the leaf-level counters. This approach changes the state machines responsible for handling read and write requests in two major ways:

- The number of extra requests necessary for validating any system data granule is limited to at most two (when neither the MAC, nor the counter can be found in the corresponding caches).
- No integrity verification is necessary when reading INs. This means that speculative execution (i.e., asynchronous verification) is limited to verification of the data MAC.

Both changes result in a lowering of the MPE complexity – no need for a full integrity tree means less logic necessary for dealing with verifications and updates to a complex data structure. The lack of a verification hierarchy also leads to more effective caching (no more high-level INs in the counter cache), more effective use of the cryptographic primitive pipelined circuits (no more need to generate MACs over INs), and shorter request queues for managing system data MACs and INs. All these characteristics apply equally well to the slightly different configuration described in the following section.

Figure 35 shows the performance impact of moving leaf counters on-chip in both loaded and unloaded test conditions. The unloaded case shows mixed results, with some benchmarks showing worse results with on-chip memory. This is most likely due to the difference in performance characteristics between the off-chip DDR4 memory and the on-chip LPDDR5 – higher base latency and smaller number of channels in the LPDDR5 can limit the performance in benchmarks sensitive to these characteristics. The results in the loaded system, however, show a large improvement, reducing the overhead over the insecure system by roughly 50% across the board.

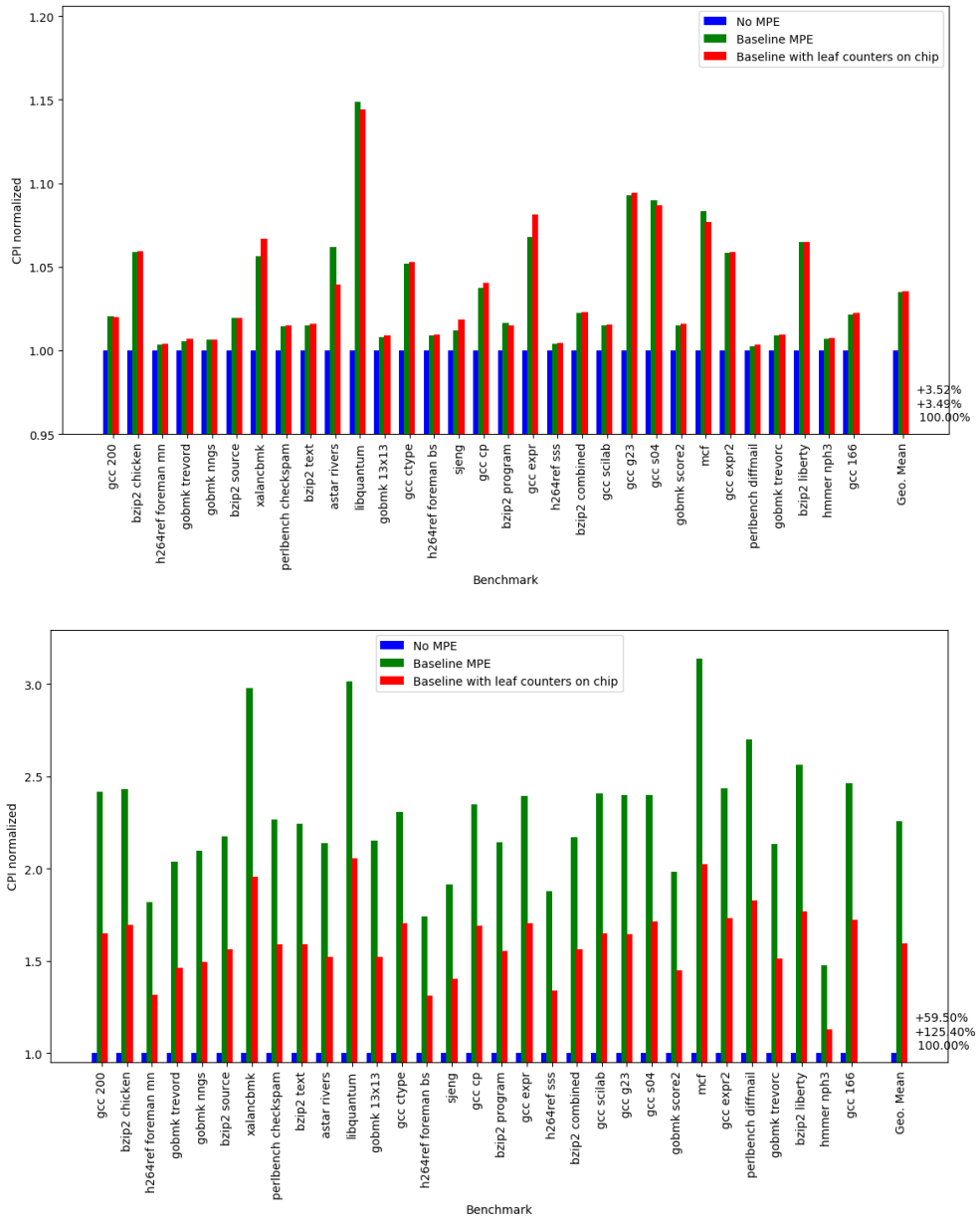

Figure 35 – Normalized CPI for a system with leaf-level counters stored on-chip, compared to the baseline MPE; above – results for unloaded system; below – results for loaded system; normalization is relative to the unprotected system

The cost of bringing the leaf-counters on-chip can be calculated proportionally to the protected memory: since each IN is 64 bytes long and covers 64 cache lines (in the baseline configuration), we need 1 byte of metadata on-chip for each 128 bytes of protected data (the size of a system cache line), so metadata would add up to $\frac{1}{128} \approx 0.78\%$ of the protected data. For 4 GiB of external DRAM this would account for 32 MiB of on-chip memory.

7.2.1.2. For tree integrity nodes

One step up from storing leaf-level counters is to store on chip the counters used to protect the leaf-level counters (henceforth called tree-level integrity nodes). The characteristics described in the previous chapter mostly apply here as well, with the temporal metadata now organised in a two-level integrity tree – one level in external memory, one found on chip.

Figure 36 shows the performance results when the tree-level integrity nodes are stored on-chip. No significant improvement can be seen in the unloaded system tests, while the loaded system tests show an improvement of roughly 13%. The large performance drop compared with the configuration with leaf-level counters on-chip can be explained through the fact that, despite having added only one level to the integrity tree, the access latency for leaf-counters is now tied to the (saturated) main memory bus.

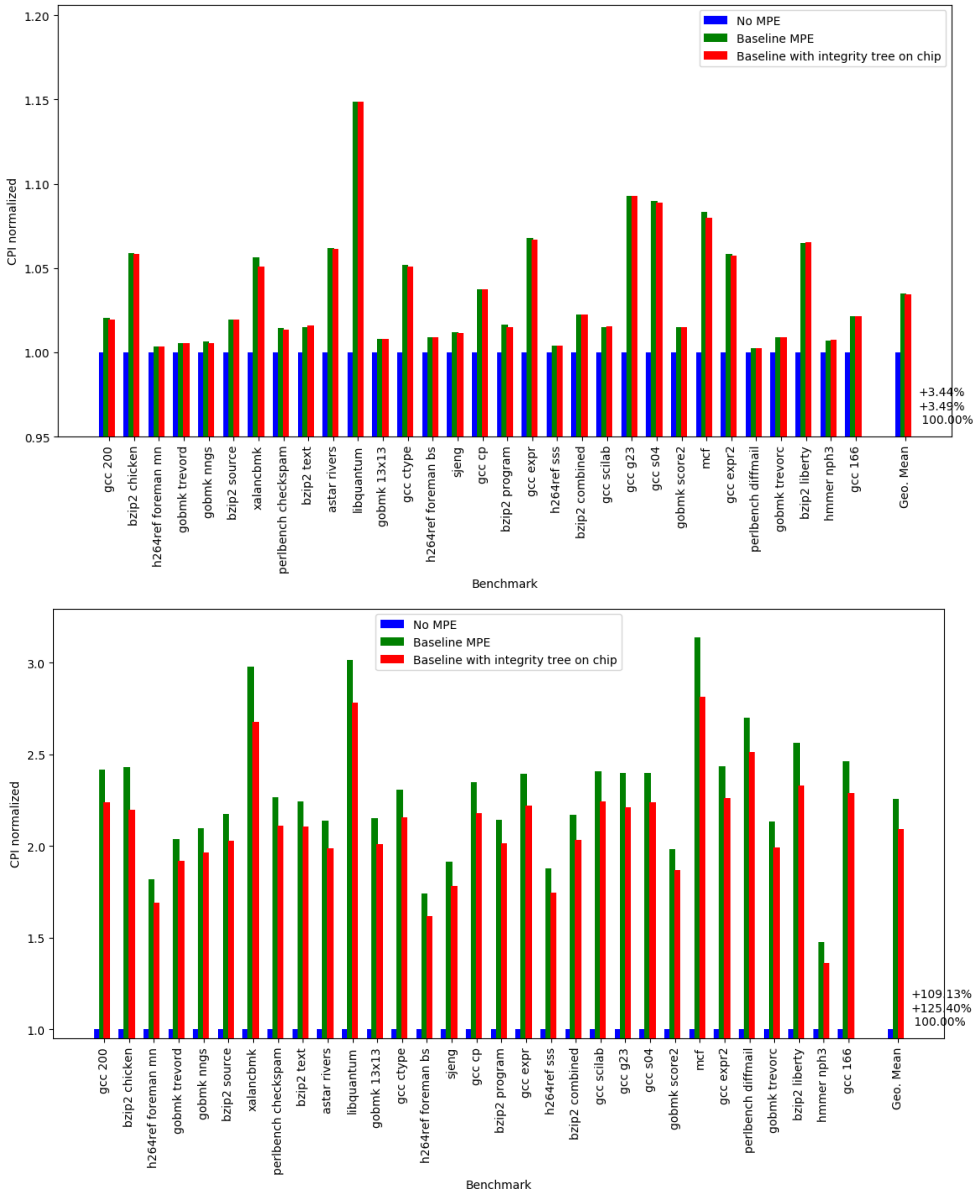

Figure 36 – Normalized CPI for a system with tree-level counters stored on-chip, compared to the baseline MPE; above – results for unloaded system; below – results for loaded system; normalization is relative to the unprotected system

The cost in memory used on-chip can be calculated similarly to that for leaf-counters: 1 byte of tree-level counters is required for each leaf-level integrity node (which is 64 bytes long), and thus relative to the whole protected memory that comes up to $\frac{1}{64} \times \frac{1}{128} = 0.012\%$. So, for 4 GiB of protected memory, 0.5 MiB of on-chip memory (half size of our L2 cache).

7.2.1.3. For system data integrity tags

The other option when it comes to relocation from off-chip to on-chip memory is to move the system data integrity tags. When stored on-chip, these tags need not be encrypted for protection, and can thus be stored as simple hashes of the corresponding data. For the sake of simplicity, we have only considered the case where the hash is computed on the ciphertext to reduce the latency of verifications on memory read. The storage of unencrypted hashes also has an impact on the state machines governing system data verification – counter values are no longer necessary to verify the integrity tag, slightly simplifying the data verification process.

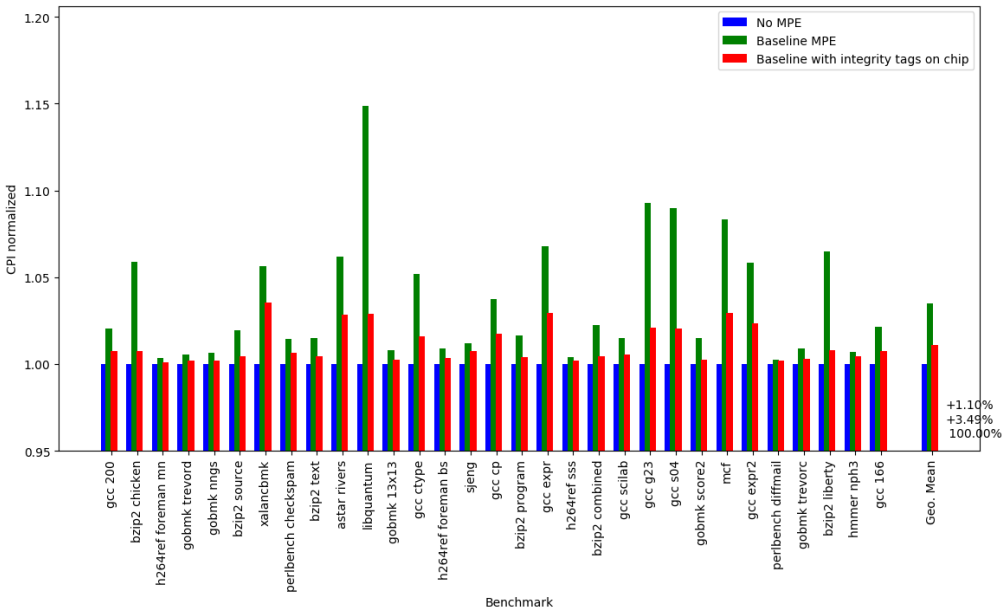

Figure 37 – Normalized CPI for an unloaded system with integrity tags for system data stored on-chip, compared to the baseline MPE; normalization is relative to the unprotected system

Figure 37 shows the performance impact of this relocation in a system with no extra traffic. When no extra memory pressure is applied, the performance improvement is outstanding, down by almost 70%. The discrepancy between this result and that presented in section 7.2.1.1 can be explained by two facts: integrity tags are more voluminous than counters, at 4 bytes per system cache line compared to 1 for counters; and the larger and denser (i.e., more memory covered per byte of cache entry) counter caches are more effective at masking the impact of IN accesses when compared to the smaller MAC cache. Shifting the bandwidth necessary to retrieve and write back MACs is thus sufficient to cause a significant performance improvement.

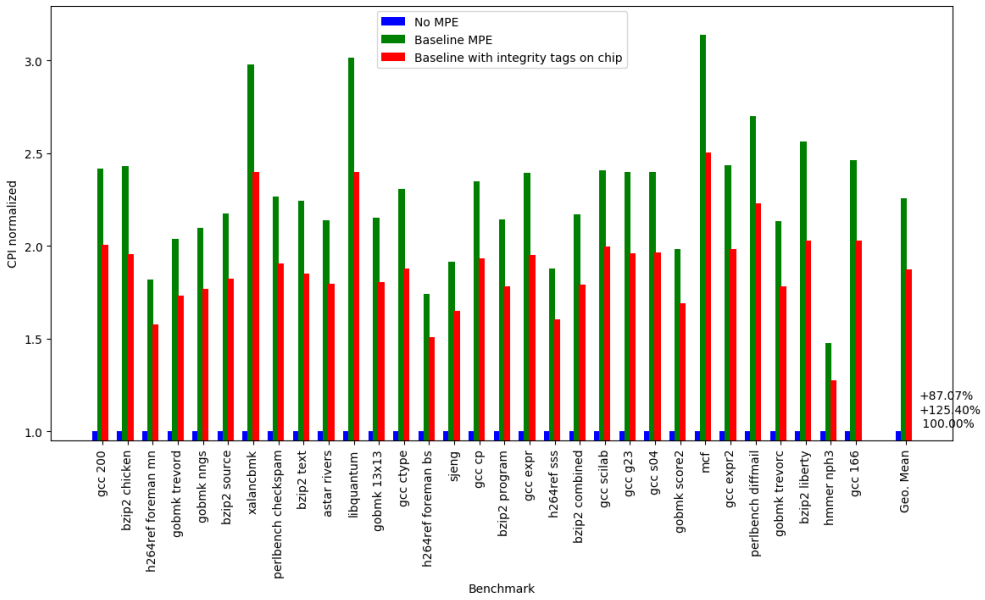

Figure 38 – Normalized CPI for a loaded system with integrity tags for system data stored on-chip, compared to the baseline MPE; normalization is relative to the unprotected system

Figure 38 shows that when the memory bus is contended, moving the integrity tags on chip still results in a large performance improvement: the normalized CPI overhead is reduced by roughly 30%. Despite being a significant improvement, it is, by comparison, less effective than relocating the integrity tree.

The cost in memory space required on chip for this relocation is of 4 bytes (the size of an integrity tag) for each system cache line of 128 bytes, so roughly $\frac{4}{128} \approx 3.13\%$ of the protected data. For 4 GiB of protected data, 128 MiB of integrity tags would be needed on-chip. Given the results in Figure 37, a cost-benefit is necessary, depending on the expected operating conditions of the device – if the device is likely to operate closer to memory bus saturation, placing 32 MiB of counters on-chip might be a better trade-off. However, in a relatively uncontended system, storing integrity tags on-chip would yield better results, though for a much larger footprint. A potential solution to this problem is the use of multi-cache-line integrity tags, which we discuss in the following section.

7.2.1.4. Multi-cache-line integrity tags and system cache integration

If the amount of space required for storing integrity tags on chip is too high, a reasonable step is to alter the relative size of integrity tags. This can be done by shortening the tags, for example, though this has an obvious impact on the security of the system by increasing the probability for an attacker to mount a successful attack. Another option is to make each integrity tag cover multiple cache lines as described in section 5.3.2.2. This, unfortunately, impacts the performance of the system by requiring all the system data associated with an integrity tag to be retrieved from memory when verifying one of the cache lines. The optimization described in section 5.3.2.2 helps mitigate this drop in performance by preventing extra reads for data that is already on chip, in the system cache.

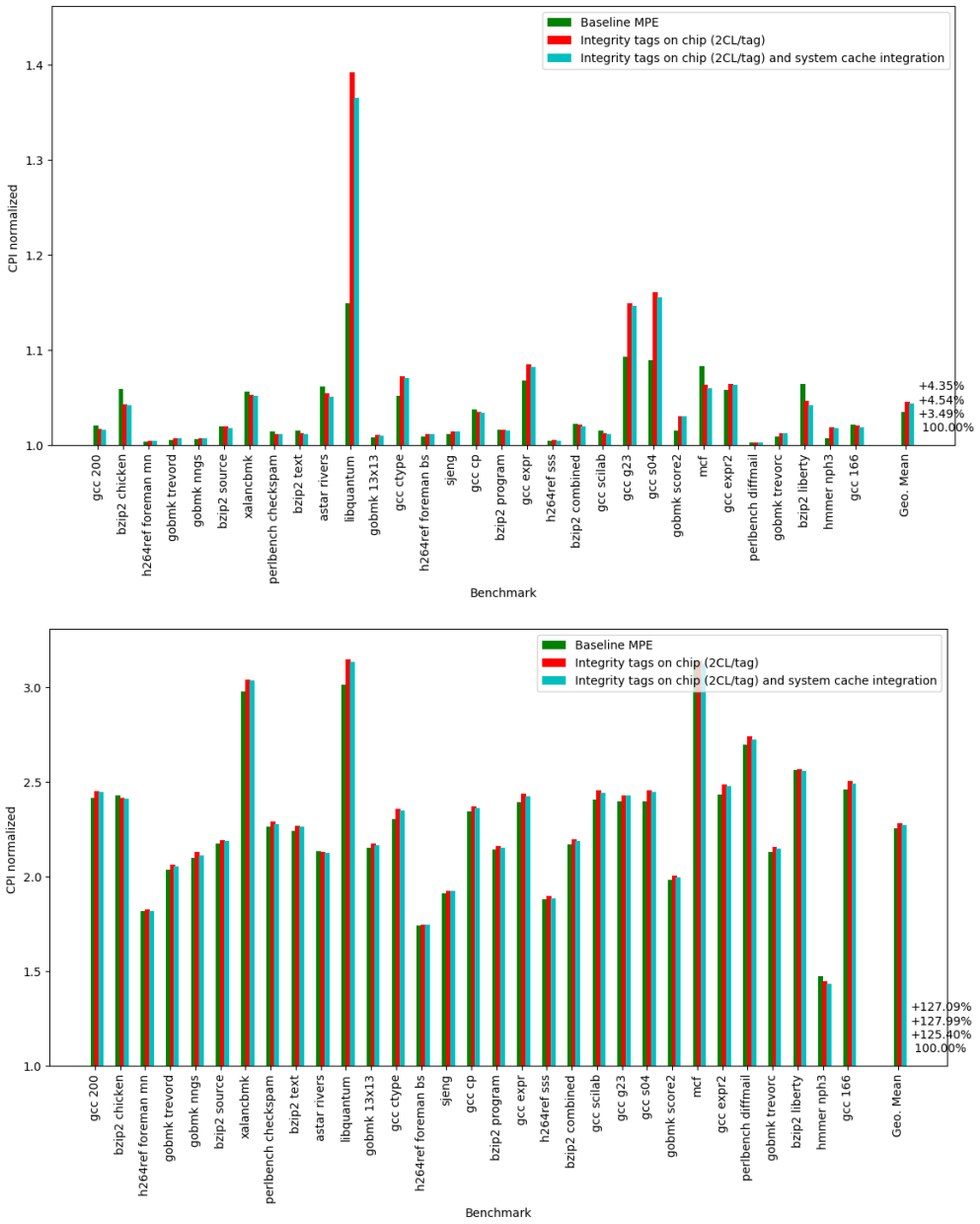

Figure 39 – Normalized CPI for a system with integrity tags covering two cache lines each stored on-chip, shown both with and without a system cache integration, compared to the baseline MPE; above – results for unloaded system; below – results for loaded system; normalization is relative to the unprotected system

Figure 39 shows results for a system configuration with 2 cache-lines per integrity tag with and without system cache integration. The results show a noticeable, but not significant, improvement when the system cache integration is available. The overall figures bring little to no improvement over the baseline MPE configuration and, even though the on-chip memory cost is only half of that in section 7.2.1.3, the lack of a performance improvement makes this configuration unappealing for deployments.

7.2.1.5. Comparing the options

The various options presented in the previous sections for MPE metadata storage on-chip showed vastly different performance profiles and costs associated with them. Table 5 summarises these results. We have omitted the configurations that included integrity tags covering more than one

cache-line per tag given their poor performance figures, focusing instead on those where each tag protects one cache line (1CL/tag).

Table 5 - Performance characteristics of the on-chip-memory configurations investigated; normalised CPI is computed relative to the unprotected system

Configuration	*Normalised CPI overhead, unloaded system*	*Normalised CPI overhead, loaded system*	*On-chip memory cost*
Baseline MPE	3.49%	125.40%	None
With leaf-counters on chip	3.52%	59.50%	0.78% of protected memory
With tree-counters on chip	3.44	109.13%	0.012% of protected memory
With integrity tags on-chip (1CL/tag)	1.10%	87.07%	3.13% of protected memory

The system data read latency reflects the performance degradation inflicted by the MPE: from the point of view of the processor, the MPE simply delays data from reaching the system cache. Since our MPE uses asynchronous verification, all that is needed for system data to be released is for the encrypted data and the encryption nonce (i.e., the corresponding counter) to be fetched from memory. Storing leaf counters on chip factors into this equation by directly reducing the latter delay, and by improving the former through a reduced load on the main memory bus. The other two optimizations have only an indirect effect: by reducing the amount of data requested from DRAM, both counter and system data read latencies are improved. Figure 41 shows a comparison between averages of these two latencies in loaded system when storing either leaf counters, tree counters, or integrity tags on chip, as well as a comparison between the average number of bytes read from DRAM. All the values are normalised relative to the averages obtained from the baseline system.

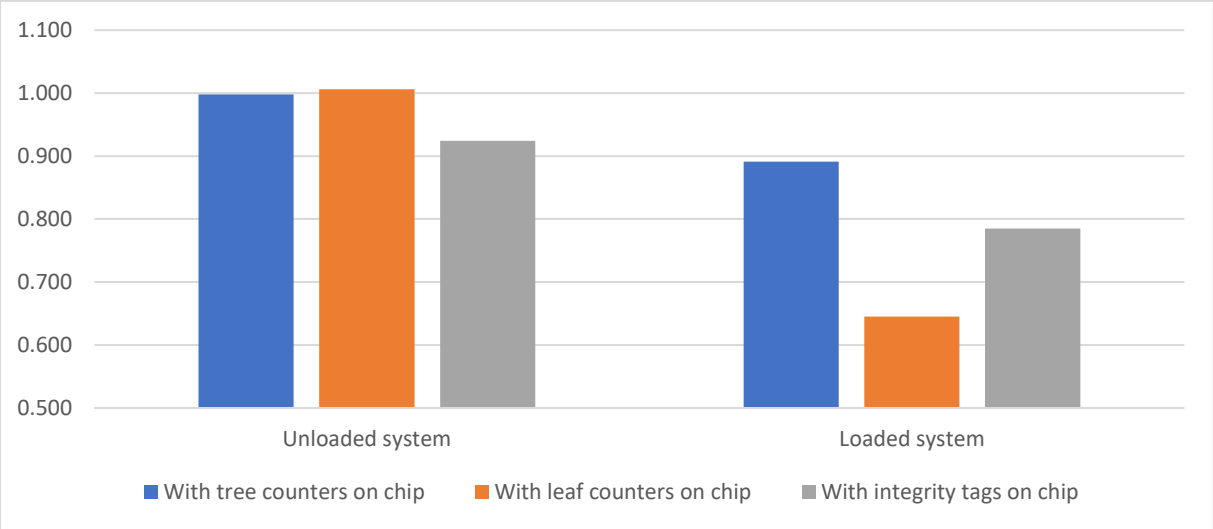

Figure 40 - Normalised system data read latency, as measured by the system cache, relative to the system with baseline MPE; the normalised values are geometric means over the entire suite of benchmarks

Relocating integrity tags from off-chip memory to on-chip memory only impacts the two shown latencies through its impact on main memory contention. By comparison, relocating the tree counters also has an impact on counter read latency through the second-order effects on the

operation of the counter logic, e.g., by modifying the occupation levels of the counter cache. Hence, the gap in counter read latency is lower than that in data read latency between the two configurations.

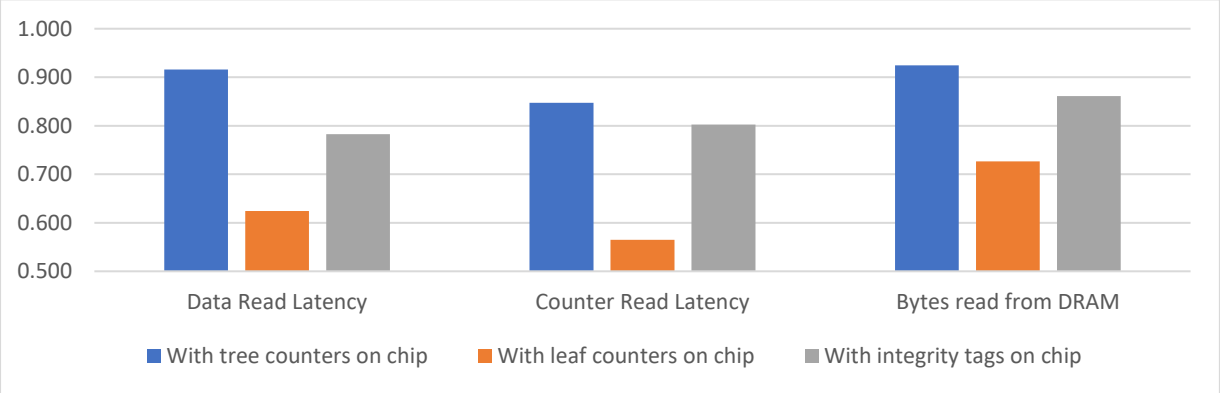

Figure 41 - Normalised performance characteristics of the on-chip-metadata configurations in a loaded system, relative to the baseline MPE; the two latency figures are as measured by the MPE with respect to its requests; the normalised values are geometric means over the entire suite of benchmarks

7.3. Assessing combined optimizations

The optimizations discussed so far have been benchmarked independently of each other to assess their independent impact on performance. Better trade-offs can, however, be reached by exploring some of the possible combinations we can create.

One appealing alternative is the use of multi-level split counters on chip – given that INs on chip need no integrity tag for protection and that our results show little performance degradation due to RMWs for configurations similar to our baseline MPE, we anticipated a configuration with 128 arity leaf-level INs on-chip could probably offer good performance. This configuration would make use of 128 3-bit minor counters, 16 4-bit middle counters, and one 64-bit major counter.

Another appealing alternative is the use of counter rebasing with simple split leaf-level counters on chip, given the effectiveness of counter rebasing on leaf-level INs.

Table 6 - Combinations of optimizations to be discussed in the following sections

Configuration	*Integrity node configuration*	*Metadata on chip*	*Dedicated memory cost*	*Other optimizations*
Multi-level split counters on chip	1 64-bit major counter, 16 4-bit middle counters, 128 3-bit minor counters	Leaf-level counters	0.4%	None
Simple split counters with rebasing on chip	1 64-bit major counter, 64 7-bit minor counters	Leaf-level counters	0.78%	Counter rebasing
Multi-level split counters and integrity tags on chip	1 64-bit major counter, 16 4-bit middle counters, 128 3-bit minor counters	Leaf-level counters and integrity tags (1CL/tag)	3.53%	None

Finally, we explored the option of storing all MPE metadata on-chip. Despite its obvious cost in space required on-chip, this configuration is poised to offer an upper limit to the performance boost the on-chip memory could bring. Table 6 gives a breakdown of these optimization combinations.

7.3.1. Multi-level split counters on chip

Given the tight budgets available for producing and running IoT devices, both in terms of cost of production and in runtime constraints such as available power, the ideal protection mechanisms would have a low footprint on the limited physical resources, as well as on the runtime performance of the platform. Two of the promising optimizations that can be combined to meet these constraints are the multi-level split counter tree and the on-chip storage. Repurposing the space previously used as integrity tag for each integrity node as middle counters, we can configure the on-chip integrity "tree" (i.e., the leaf-counters only) to an arity of 128, as described in Table 6. This cuts down to half the size required for the on-chip memory from the baseline value, to roughly 0.4% of the protected memory – roughly 16 MiB for 4 GiB of protected external DRAM. At the same time, the overhead of RMWs produced by 3-bit minor counters is kept in check by the grouping afforded by middle counters.

Figure 42 gives the results for both loaded and unloaded systems using this configuration. A minimal performance drop can be seen for the unloaded system, compared to the baseline configuration. For the loaded system this configuration yields only half the performance degradation of the baseline configuration. Compared to the simple split leaf-level counters on chip, slightly worse numbers are achieved both in an unloaded system (3.6% vs 3.52%), and in a loaded one (62.26% vs 59.50%). The difference is, however, negligible compared with the benefit of halving the on-chip required memory size.

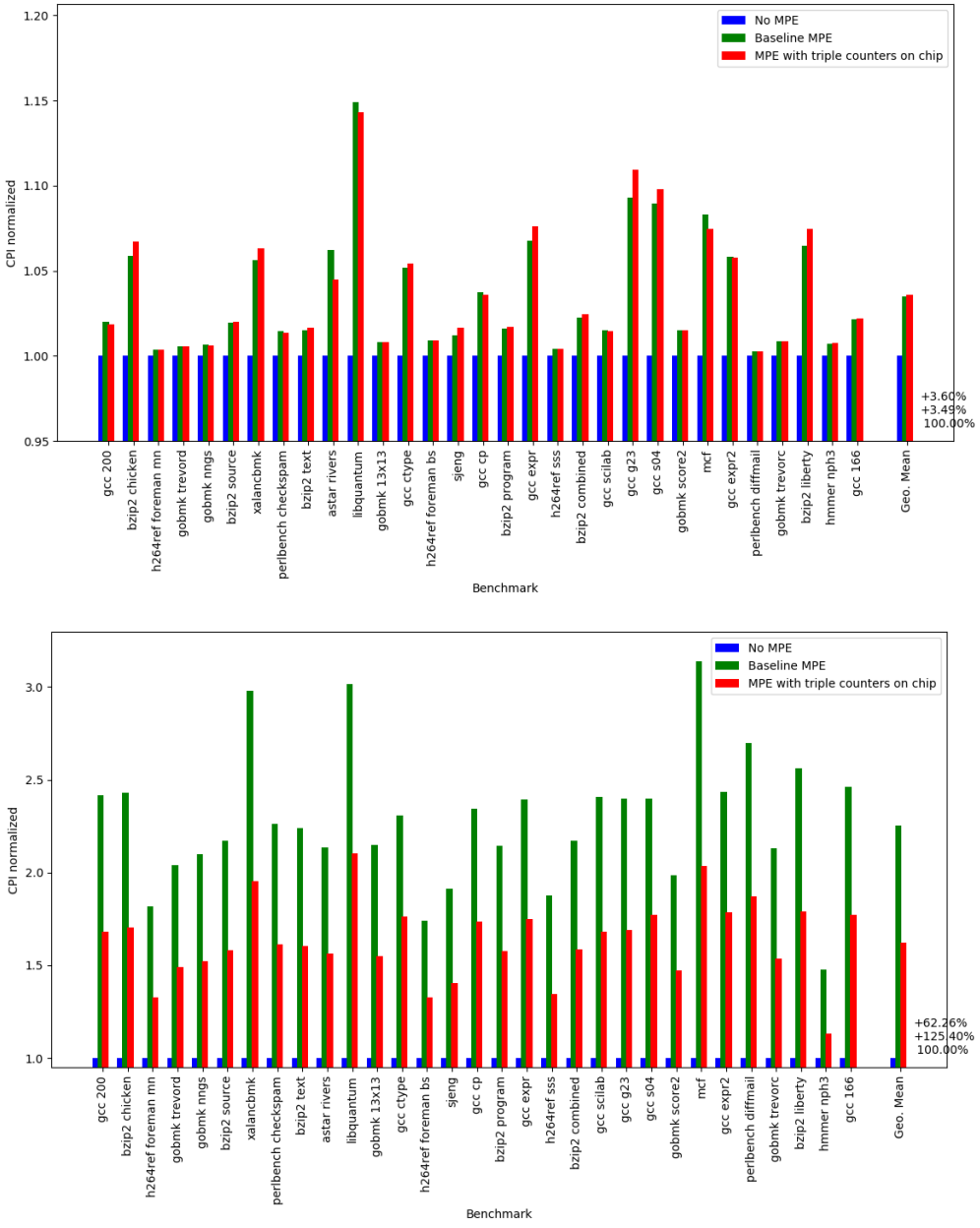

Figure 42 – Normalized CPI for a system with triple leaf-level counters stored on-chip compared to the baseline MPE, relative to unprotected system; above – results for unloaded system; below – results for loaded system

7.3.2. Leaf-level counters with rebasing on chip

Another optimization from section 7.1 which promising results was that of counter rebasing. While the actual CPI figures showed little improvement with rebasing, the number of RMW operations saw a significant drop, particularly at leaf-level. Since moving the integrity tree on chip leaves only the leaf-level counters, we speculated that the drop in RMWs could lead to noticeable improvements in the loaded system. Figure 43 shows the results which, unfortunately, show no improvement over the leaf-counters-on-chip configuration.

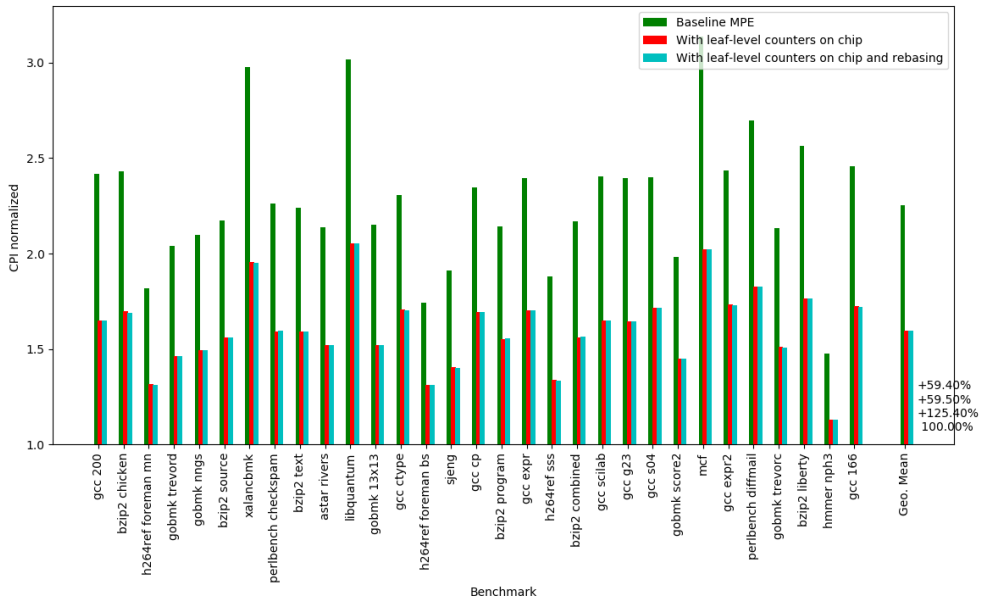

Figure 43 – Normalized CPI for a loaded system with leaf-level counters stored on-chip with and without rebasing compared to the baseline MPE; normalization was performed relative to the unprotected system

7.3.3. Multi-level split counters and integrity tags on chip

We have investigated the effect of relocating counters and integrity tags to on-chip memory, and both have yielded significant improvements. However, it is also worth experimenting with having all MPE metadata on chip as a point of comparison. Given the promising results from section 7.3.1, the configuration includes triple counters instead of the simple split counters found in the baseline configuration.

While it is tempting to assume the two performance improvements will combine to yield barely any overhead even in the loaded system case, this configuration has a significant drawback: since all the MPE traffic has moved from one memory bus to another, it is quite possible that the on-chip memory bus becomes a performance bottleneck in the same way that the main memory bus is a bottleneck for the baseline system. Whether this happens is obviously down to the characteristics of the on-chip memory – in our case, the single-channel LPDDR module is rather limited in its maximum supported bandwidth. If technologies such as HBM become affordable enough for deployment on IoT devices, this perspective could very well change. Figure 44 shows the performance figures for this configuration in both loaded and unloaded scenarios.

The results point to a degree of saturation on the on-chip memory bus – the overhead in the loaded system shows a drop from the results for leaf-level counters alone (53.69% vs 59.50%), but remains high overall, at over 50% of the unprotected system. To confirm our previous theory, we can look at the two latencies discussed in section 7.2.1.5 – data read latency and counter read latency (Figure 45). Data read latency can be seen to drop significantly in this configuration, while counter read latency suffers a large increase. This spike in latency for reads from on-chip memory could also turn problematic if this memory is also shared by other hardware or software components running on the platform.

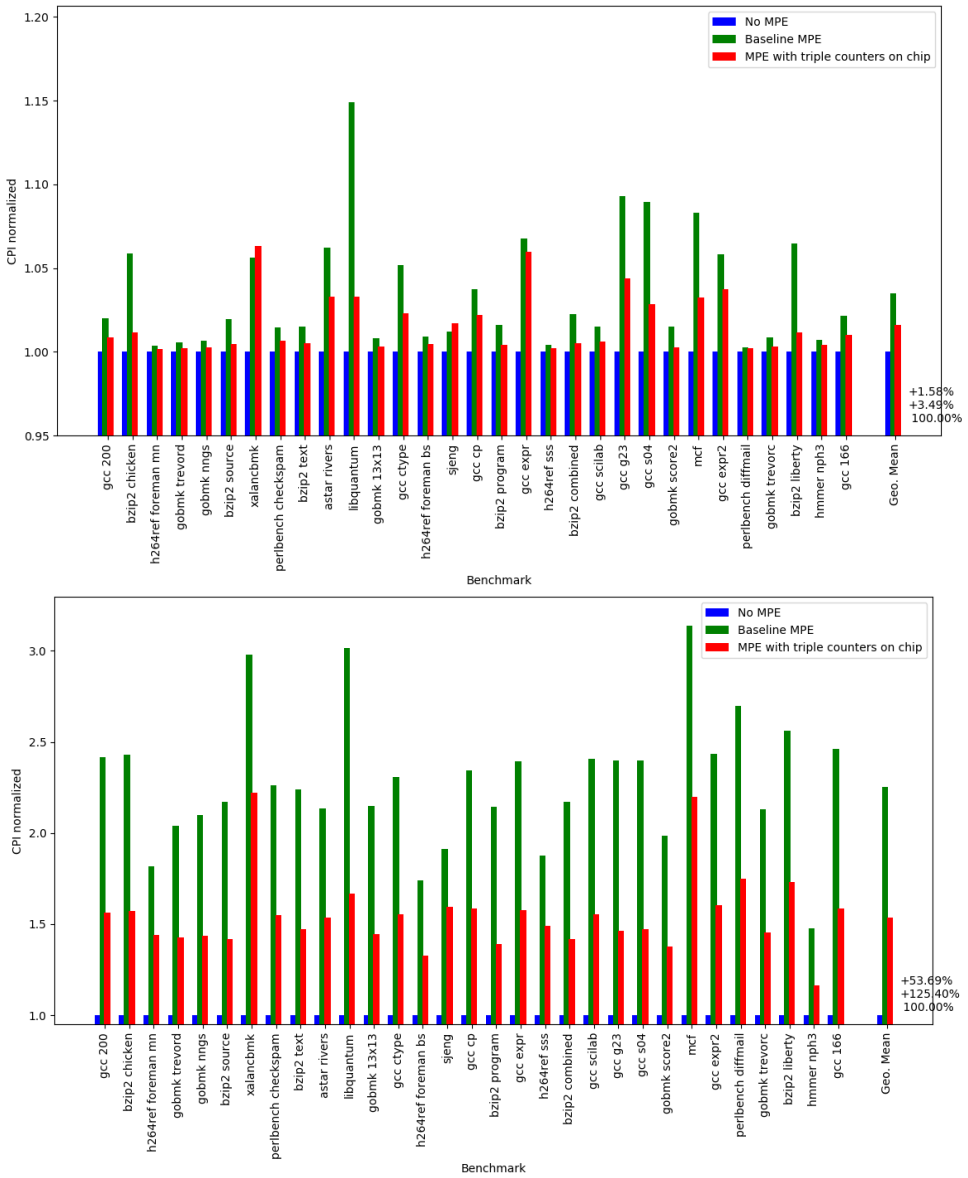

Figure 44 – Normalized CPI for a system with leaf-level counters and integrity tags stored on-chip compared to the baseline MPE, relative to unprotected system; above – results for unloaded system; below – results for loaded system

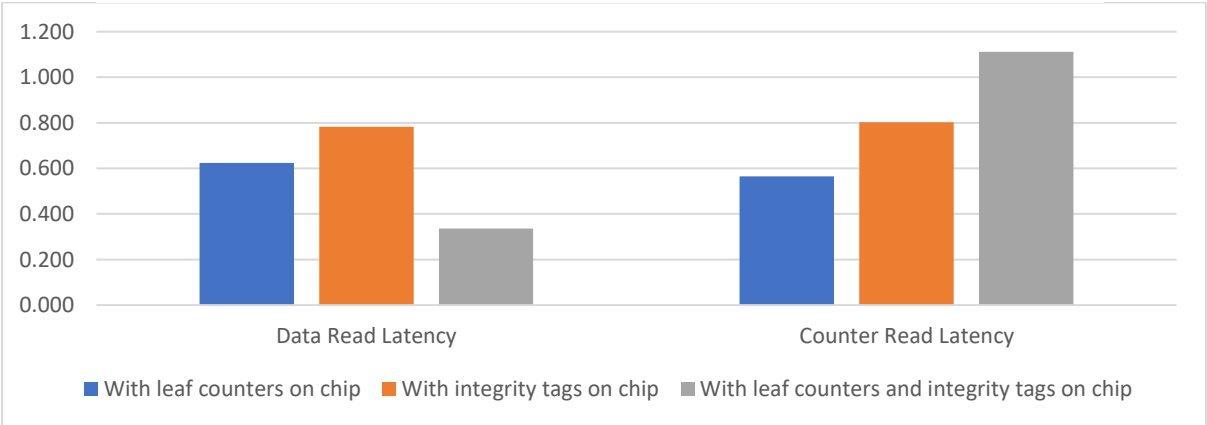

Figure 45 - Normalised data and counter read latencies of on-chip-metadata configurations in a loaded system, relative to the baseline MPE; the two figures are as measured by the MPE with respect to its requests; the normalised values are geometric means over all the benchmarks

8. Conclusion

This chapter concludes the thesis, looking back at how the work has aligned with our objectives, and glancing forward to other promising avenues for memory protection.

8.1. Project objectives

The aim for this thesis has been centred around physical memory protection for IoT devices. The objectives that we presented in section 1.2.2 have all been tackled:

- *To establish the categories of IoT devices that might benefit from memory protection against physical attacks*: the taxonomy of IoT devices has been discussed in chapter 2, along with a breakdown of the categories most likely to benefit from our techniques. Current best practices indicate a need for more security against physical attacks for exposed platforms for which such attacks can be facile (such as IoT devices). We have made the case, however, that given the relatively low performance profile of most IoT devices, only higher-end devices could realistically see a strong case for cryptographic memory protection, mainly due to its high cost.
- *To set forth IoT-specific characteristics based on which security mechanisms can be evaluated*: while multiple traits can serve as metrics for security mitigation functionality, we have argued in section 2.5 that performance degradation is our main concern due to its role in slowing down adoption. The memory cost of the metadata required for our techniques was also relevant and considered in choosing which techniques to investigate, not least because of the intricate relationship between these two characteristics. Sections 7.2 and 7.3 in particular represent an attempt at achieving the best possible balance in this trade-off.
- *To present the need for and scope of cryptographic memory protection*: chapter 2 makes the case that physical exposure is part of the basic conditions for a vast number of IoT platform, which opens them up for special categories of attacks. Chapter **Error! Reference source not found.** then goes into the details of these physical attack avenues of most concern to our area of investigation. As a counter to these attacks, a short taxonomy of mitigations aimed at reducing their feasibility is also presented. The scope of our investigation is set in section 3.2 as security goals to be achieved by the mechanisms we propose and prototype.
- *To identify and analyse novel integrity and replay protection algorithms and techniques, based on the relevant characteristics*: before novel techniques could be presented, we set the stage in chapter 4 by providing a breakdown and review of the cryptographic algorithms involved in offering our security guarantees. However, achieving the goals highlighted in 3.2 is only half the solution: having these mitigations deployed on real platforms require them to have reasonable performance characteristics. Section 4.2 thus goes on to explore the space of existing performance enhancements to the basic mechanisms presented in 4.1. Chapter 5 then presents our contributions to improving the performance of cryptographic memory protection, both in terms of new techniques, and in previously proposed techniques that we have implemented on top of a baseline prototype.
- *To prototype and evaluate these algorithms to determine their feasibility in the case of different classes of IoT devices*: the mechanisms discussed in chapter 5 represent optimizations to be brought to an existing system that offers cryptographic memory protection. We have therefore built these optimizations on a baseline system taken from the research done in [4]. Chapter 6 dives into the architecture of the prototyping framework, of

the baseline system described previously, and discusses the way we benchmark and assess our prototypes. Chapter 7 then provides the benchmarking results and discusses their relative merit in improving the performance of our system.

The success of our proposed contributions has been mixed – results presented in chapter 7 show little improvement on any metric from some techniques (e.g., system cache integration usage for reducing the number of RMW operations), while others provided significant improvements. To summarise the most notable results:

- Counter rebasing is particularly effective at reducing the number of minor counter overflows at leaf level in the integrity tree (section 7.1.4).
- Storing leaf-level counters in a dedicated LPDDR PoP memory enclosed in the same package as the CPU can greatly improve the performance of the MPE on heavily-loaded systems, at a relatively small cost in on-chip memory (section 7.2.1.1).
- Multi-level counters can be used to reduce to half the amount of space required in the dedicated on-chip memory when storing leaf-level counters, without incurring a major performance drop (section 7.3.1).

8.2. Future directions

The optimizations discussed in this thesis represent only a small step towards making cryptographic memory protection feasible for IoT platforms. At 50% average performance overhead on a loaded system, which goes up towards 100% for specific types of applications, our model would most likely be rejected for everything but the most security-conscious platforms. Some potential future investigation directions include:

- **Integrity tag compression**: results in this thesis and in previous works shows that integrity tags represent a major source of performance degradation due to their density (4 bytes per system data cache line, in our case). While the performance hit can be reduced by relocating the tags to on-chip memory (as shown in section 7.2.1.3), the cost in on-chip memory is significant, especially when compared with a similar relocation of the integrity tree instead. Compressing these tags by covering a larger amount of system data with each one proved counterproductive due to its other implications for the system (section 7.2.1.4). More clever compression techniques could be consequential in bringing the performance overhead to reasonable levels.
- **Dynamic integrity nodes on chip**: our investigation in utilising dedicated on-chip memory only focused on static integrity trees. In recent years, however, multiple dynamic designs have been proposed and shown to perform particularly well. While skewed trees [95] would not be useful if the entire concept of an integrity *tree* is set aside (as when the integrity nodes are found on chip), designs such as MorphCounter [84] could yield even better compression of counters on chip.

A separate direction aimed at improving the accuracy of our results could include changes to the benchmarking techniques:

- **More realistic setup**: our numbers have been obtained in a single-core setup. Many IoT platforms have multiple cores sharing the primary memory – results collected with more representative, multi-threaded benchmarks on a multi-core system setup would inspire more confidence in their accuracy. At the same time, our results have been obtained running either a lone benchmark, or a benchmark and synthetic traffic generators. Neither

of these options is realistic and could be improved by emitting traffic that more closely resembles the activity of a processor.

- **More representative benchmarks**: despite containing a wide array of workload types, the SPEC 2006 benchmark suite is likely not representative enough of software running on IoT platforms. Replacing these traces with ones derived from IoT-specific benchmark suites [104] could improve result accuracy.

Another topic which could be investigated is that of protecting non-volatile memories and the degree to which our techniques are relevant in their context.

References

[1] S. Li, L. Da Xu and S. Zhao, “The internet of things: a survey,” *Information Systems Frontiers,* vol. 17, p. 243–259, 2015.

[2] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey and others, “The matter of heartbleed,” in *Proceedings of the 2014 conference on internet measurement conference*, 2014.

[3] J. Götzfried and T. Müller, “Analysing Android's Full Disk Encryption Feature.,” *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.,* vol. 5, p. 84–100, 2014.

[4] D. Schall, “Evaluation and Optimization of Memory Encryption and Integrity Protection. Master Thesis,” University of Kaiserslautern, Department of Electrical Engineering and Information Technology, Microelectronic Systems Design Research Group, 2019.

[5] C. Bormann, M. Ersue and A. Keranen, “Terminology for constrained-node networks,” *Internet Engineering Task Force (IETF): Fremont, CA, USA,* p. 2070–1721, 2014.

[6] RaspberryPi, “Raspberry Pi 4 Tech Specs”.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, “Edge computing: Vision and challenges,” *IEEE internet of things journal,* vol. 3, p. 637–646, 2016.

[8] Arm, *Arm Roadshow Slides,* 2018.

[9] M. O. Ojo, S. Giordano, G. Procissi and I. N. Seitanidis, “A review of low-end, middle-end, and high-end IoT devices,” *IEEE Access,* vol. 6, p. 70528–70554, 2018.

[10] Sidhartha, “Classification of Semiconductor Memories and Computer Memories,” 2015. [Online]. Available: https://www.vlsifacts.com/classification-of-semiconductor-memories-and-computer-memories/. [Accessed 12 January 2022].

[11] O. Kömmerling and M. G. Kuhn, “Design Principles for Tamper-Resistant Smartcard Processors.,” *Smartcard,* vol. 99, p. 9–20, 1999.

[12] C. Visinescu, *Why IoT Security Matters,* 2021.

[13] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu and D. Qiu, “Security of the Internet of Things: perspectives and challenges,” *Wireless Networks,* vol. 20, p. 2481–2501, 2014.

[14] S. Babar, P. Mahalle, A. Stango, N. Prasad and R. Prasad, “Proposed security model and threat taxonomy for the Internet of Things (IoT),” in *International Conference on Network Security and Applications*, 2010.

[15] ENISA, “GOOD PRACTICES FOR SECURITY OF IOT,” https://www.enisa.europa.eu/publications/good-practices-for-security-of-iot-1, 2019.

[16] D. Miessler, “Securing the internet of things: Mapping attack surface areas using the OWASP IoT top 10,” in *RSA Conference*, 2015.

[17] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom and R. Strackx, “Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution,” in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018.

[18] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun and A.-R. Sadeghi, “Software grand exposure:{SGX} cache attacks are practical,” in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[19] M. Schwarz, S. Weiser, D. Gruss, C. Maurice and S. Mangard, “Malware guard extension: Using SGX to conceal cache attacks,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[20] A. Nilsson, P. N. Bideh and J. Brorsson, “A survey of published attacks on Intel SGX,” *arXiv preprint arXiv:2006.13598,* 2020.

[21] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald and F. D. Garcia, “VoltPillager: Hardware-based fault injection attacks against Intel {SGX} Enclaves using the {SVID} voltage scaling interface,” in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[22] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai and R. A. Popa, “An off-chip attack on hardware enclaves via the memory bus,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[23] J. Crenne, R. Vaslin, G. Gogniat, J.-P. Diguet, R. Tessier and D. Unnikrishnan, “Configurable memory security in embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS),* vol. 12, p. 1–23, 2013.

[24] M. Miller, “Trends, challenge, and shifts in software vulnerability mitigation, 2019,” *URL https://github. com/microsoft/MSRC-Security-Research/raw/master/presentations/2019_{0}{2}{_{B}{l}{u}}eHatIL/2019_{0}{1}.*

[25] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher and others, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[26] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, “Meltdown,” *arXiv preprint arXiv:1801.01207,* 2018.

[27] M. G. Kuhn, “Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP,” *IEEE Transactions on Computers,* vol. 47, p. 1153–1157, 1998.

[28] Z. Cohen, “Implementing Physical Layer Security in IoT Devices Using Additive Manufacturing,” 2019. [Online]. Available: https://www.nano-di.com/blog/2019-implementing-physical-layer-security-in-iot-devices-using-additive-manufacturing. [Accessed 12 January 2022].

[29] A. Trikalinou and D. Lake, “Taking DMA attacks to the next level,” *BlackHat USA,* 2017.

[30] T. Lecroy, “DDR Memory Testing Part II: Using Interposers,” 01 October 2014. [Online]. Available: https://blog.teledynelecroy.com/2014/10/ddr-memory-testing-part-ii-using.html. [Accessed 27 February 2022].

[31] Y.-S. Won, S. Chatterjee, D. Jap, A. Basu and S. Bhasin, “DeepFreeze: Cold Boot Attacks and High Fidelity Model Recovery on Commercial EdgeML Device,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[32] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum and E. W. Felten, “Lest we remember: cold-boot attacks on encryption keys,” *Communications of the ACM,* vol. 52, p. 91–98, 2009.

[33] S. F. Yitbarek, M. T. Aga, R. Das and T. Austin, “Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[34] E.-O. Blass and W. Robertson, “TRESOR-HUNT: attacking CPU-bound encryption,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[35] U. Frisk, “macOS FileVault2 Password Retrieval,” Security | DMA | Hacking, 15 December 2016. [Online]. Available: https://blog.frizk.net/2016/12/filevault-password-retrieval.html. [Accessed 28 February 2022].

[36] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors,” *ACM SIGARCH Computer Architecture News,* vol. 42, p. 361–372, 2014.

[37] S. Checkoway and H. Shacham, “Iago attacks: Why the system call API is a bad untrusted RPC interface,” *ACM SIGARCH Computer Architecture News,* vol. 41, p. 253–264, 2013.

[38] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson and E. Encrenaz, “Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller,” in *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2013.

[39] A. Barenghi, L. Breveglieri, I. Koren and D. Naccache, “Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures,” *Proceedings of the IEEE,* vol. 100, p. 3056–3076, 2012.

[40] L. Zussa, J.-M. Dutertre, J. Clédiere, B. Robisson, A. Tria and others, “Investigation of timing constraints violation as a fault injection means,” in *27th Conference on Design of Circuits and Integrated Systems (DCIS), Avignon, France*, 2012.

[41] S. Skorobogatov, “How microprobing can attack encrypted memory,” in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017.

[42] F.-X. Standaert, “Introduction to side-channel attacks,” in *Secure integrated circuits and systems*, Springer, 2010, p. 27–42.

[43] R. Hund, C. Willems and T. Holz, “Practical timing side channel attacks against kernel space ASLR,” in *2013 IEEE Symposium on Security and Privacy*, 2013.

[44] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A High Resolution, Low Noise, L3 Cache {Side-Channel} Attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014.

[45] D. Gruss, C. Maurice, K. Wagner and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[46] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice and S. Mangard, "{ARMageddon}: Cache Attacks on Mobile Devices," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[47] E. Dubrova, "Anti-tamper techniques," *KTH Royal Institute of Technology, Sweden,* 2018.

[48] S. Ravi, A. Raghunathan and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *17th International Conference on VLSI Design. Proceedings.*, 2004.

[49] R. N. Akram, K. Markantonakis and K. Mayes, "User centric security model for tamper-resistant devices," in *2011 IEEE 8th International Conference on e-Business Engineering*, 2011.

[50] E. Peterson, *Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices,* Aug, 2018.

[51] K. Garb, J. Obermaier, E. Ferres and M. Künig, "FORTRESS: FORtified Tamper-Resistant Envelope with Embedded Security Sensor," in *2021 18th International Conference on Privacy, Security and Trust (PST)*, 2021.

[52] A. Srivastava and P. Ghosh, "A Novel Approach of Data Content Zeroization Under Memory Attacks," *Journal of Electronic Testing,* vol. 36, p. 147–167, 2020.

[53] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni and G. Alonso, "StRoM: smart remote memory," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[54] S. Swami, J. Rakshit and K. Mohanram, "STASH: Security architecture for smart hybrid memories," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[55] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," *ACM SIGARCH Computer Architecture News,* vol. 45, p. 94–106, 2017.

[56] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009.

[57] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM Journal on computing,* vol. 43, p. 831–871, 2014.

[58] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2015.

[59] S. Gao, "Efficient fully homomorphic encryption scheme," *Cryptology ePrint Archive,* 2018.

[60] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017.

[61] J. T. Pawlowski, “Hybrid memory cube (HMC),” in *2011 IEEE Hot chips 23 symposium (HCS)*, 2011.

[62] S. Swami and K. Mohanram, “ARSENAL: Architecture for secure non-volatile memories,” *IEEE Computer Architecture Letters,* vol. 17, p. 192–196, 2018.

[63] P. Zuo, Y. Hua, M. Zhao, W. Zhou and Y. Guo, “Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[64] S. Swami and K. Mohanram, “ACME: Advanced counter mode encryption for secure non-volatile memories,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[65] J. Daemen and V. Rijmen, “Reijndael: The Advanced Encryption Standard.,” *Dr. Dobb's Journal: Software Tools for the Professional Programmer,* vol. 26, p. 137–139, 2001.

[66] L. Martin, “XTS: A mode of AES for encrypting hard disks,” *IEEE Security & Privacy,* vol. 8, p. 68–69, 2010.

[67] D. Kaplan, J. Powell and T. Woller, “AMD memory encryption,” *White paper,* 2016.

[68] S. Gueron, *A Memory Encryption Engine Suitable for General Purpose Processors.(2016),* 2016.

[69] M. Liskov, R. L. Rivest and D. Wagner, “Tweakable block ciphers,” in *Annual International Cryptology Conference*, 2002.

[70] R. Avanzi, “The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes,” *IACR Transactions on Symmetric Cryptology,* p. 4–44, 2017.

[71] V. Frascino, “ARM v8. 5 Memory Tagging Extension,” in *Proceedings of the Linux Plumbers Conference, Lisbon, Portugal*, 2019.

[72] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger and others, “PRINCE–a low-latency block cipher for pervasive computing applications,” in *International Conference on the Theory and Application of Cryptology and Information Security*, 2012.

[73] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita and F. Regazzoni, “Midori: A block cipher for low energy,” in *International Conference on the Theory and Application of Cryptology and Information Security*, 2015.

[74] J. L. Carter and M. N. Wegman, “Universal classes of hash functions,” *Journal of computer and system sciences,* vol. 18, p. 143–154, 1979.

[75] D. McGrew and J. Viega, “The Galois/counter mode of operation (GCM),” *submission to NIST Modes of Operation Process,* vol. 20, p. 0278–0070, 2004.

[76] R. Avanzi, S. Banik, A. Bogdanov, O. Dunkelman, S. Huang and F. Regazzoni, “Qameleon v. 1.0,” *A Submission to the NIST Lightweight Cryptography Standardization Process,* 2019.

[77] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk and S. Devadas, “AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing,” in *ACM International Conference on Supercomputing 25th Anniversary Volume*, New York, NY, USA, 2003.

[78] S. Chhabra, B. Rogers, Y. Solihin and M. Prvulovic, “SecureME: a hardware-software approach to full system security,” in *Proceedings of the international conference on Supercomputing*, 2011.

[79] S. Chhabra, Towards Performance, System and Security Issues in Secure Processor Architectures, North Carolina State University, 2010.

[80] B. Gassend, E. Suh, D. Clarke, M. Van Dijk and S. Devadas, “Caches and merkle trees for efficient memory authentication,” in *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, 2003.

[81] C. Yan, D. Englender, M. Prvulovic, B. Rogers and Y. Solihin, “Improving cost, performance, and security of memory encryption and authentication,” *ACM SIGARCH Computer Architecture News,* vol. 34, p. 179–190, 2006.

[82] J. Yang, Y. Zhang and L. Gao, “Fast secure processor for inhibiting software piracy and tampering,” in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003.

[83] M. Taassori, A. Shafiee and R. Balasubramonian, “VAULT: Reducing paging overheads in SGX with efficient integrity verification structures,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[84] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao and M. K. Qureshi, “Morphable counters: Enabling compact integrity trees for low-overhead secure memories,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[85] J. Lee, T. Kim and J. Huh, “Reducing the memory bandwidth overheads of hardware security support for multi-core processors,” *IEEE Transactions on Computers,* vol. 65, p. 3384–3397, 2016.

[86] W. E. Hall and C. S. Jutla, “Parallelizable authentication trees,” in *International Workshop on Selected Areas in Cryptography*, 2005.

[87] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli and P. Guillemin, “Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2007.

[88] T. S. Lehman, A. D. Hilton and B. C. Lee, “PoisonIvy: Safe speculation for secure memory,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[89] W. Shi, H.-h. S. Lee, M. Ghosh, C. Lu and A. Boldyreva, “High efficiency counter mode security architecture via prediction and precomputation,” in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

[90] W. Shi and H.-H. S. Lee, “Accelerating memory decryption and authentication with frequent value prediction,” in *Proceedings of the 4th international conference on Computing frontiers*, 2007.

[91] B. Rogers, S. Chhabra, M. Prvulovic and Y. Solihin, “Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly,” in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[92] J. Szefer and S. Biedermann, “Towards fast hardware memory integrity checking with skewed merkle trees,” in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014.

[93] S. Vig, T. Y. Tzer, G. Jiang and S.-K. Lam, “Customizing skewed trees for fast memory integrity verification in embedded systems,” in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017.

[94] S. Vig, G. Jiang and S.-K. Lam, “Dynamic skewed tree for fast memory integrity verification,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[95] S. Vig, R. Juneja, G. Jiang, S.-K. Lam and C. Ou, “Framework for fast memory authentication using dynamically skewed integrity tree,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 27, p. 2331–2343, 2019.

[96] S. Vig, R. Juneja and S.-K. Lam, “DISSECT: dynamic skew-and-split tree for memory authentication,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020.

[97] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein and others, “Measuring the impact of spectre and meltdown,” in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018.

[98] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti and others, “The gem5 simulator,” *ACM SIGARCH computer architecture news,* vol. 39, p. 1–7, 2011.

[99] Arm, “Cortex A57”.

[100] Matplotlib, “Matplotlib,” 2021. [Online]. Available: https://matplotlib.org/. [Accessed 12 January 2022].

[101] NumPy, “NumPy,” 2022. [Online]. Available: https://numpy.org/. [Accessed 12 January 2022].

[102] SPEC, *SPEC CPU 2006,* 2006.

[103] SPEC, *SPEC CPU 2017,* 2017.

[104] C. A. Boano, S. Duquennoy, A. Förster, O. Gnawali, R. Jacob, H.-S. Kim, O. Landsiedel, R. Marfievici, L. Mottola, G. P. Picco and others, “IoTBench: Towards a benchmark for low-power wireless networking,” in *2018 IEEE Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench)*, 2018.

[105] A. Das, S. Patterson and M. Wittie, “Edgebench: Benchmarking edge computing platforms,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.

[106] A. Sandberg, “Understanding Multicore Performance: Efficient Memory System Modeling and Simulation. Doctoral Thesis,” Uppsala University, Disciplinary Domain of Science and Technology, Mathematics and Computer Science, Department of Information Technology, Division of Computer Systems, Uppsala, 2014.

[107] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, “Automatically characterizing large scale program behavior,” *ACM SIGPLAN Notices,* vol. 37, p. 45–57, 2002.

[108] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard and A.-R. Sadeghi, “TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V.,” in *NDSS*, 2019.

[109] S. Volos, K. Vaswani and R. Bruno, “Graviton: Trusted execution environments on gpus,” in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018.

[110] J. Szefer and R. B. Lee, “Architectural support for hypervisor-secure virtualization,” *ACM SIGPLAN Notices,* vol. 47, p. 437–450, 2012.

[111] H. Suo, J. Wan, C. Zou and J. Liu, “Security in the Internet of Things: A Review,” in *2012 International Conference on Computer Science and Electronics Engineering*, 2012.

[112] P. S. Ramrakhyani, R. Avanzi and W. A. Elsasser, *Counter integrity tree for memory security,* Google Patents, 2019.

[113] Lunkwill, *Tux secure,* 2006.

[114] Lunkwill, *Tux ecb,* 2006.

[115] H. Lipmaa, P. Rogaway and D. Wagner, “CTR-mode encryption,” in *First NIST Workshop on Modes of Operation*, 2000.

[116] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” *SIGPLAN Not.,* vol. 35, p. 168–177, 11 2000.

[117] D. Karnazes, *Embedded System Security for Hardware Designers,* 2020.

[118] S. Jin, J. Ahn, S. Cha and J. Huh, “Architectural support for secure virtualization under a vulnerable hypervisor,” in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[119] J. Gubbi, R. Buyya, S. Marusic and M. Palaniswami, “Internet of Things (IoT): A vision, architectural elements, and future directions,” *Future generation computer systems,* vol. 29, p. 1645–1660, 2013.

[120] C. Göttel, R. Pires, I. Rocha, S. Vaucher, P. Felber, M. Pasin and V. Schiavoni, “Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms,” in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[121] L. Ewing, *Tux,* 1996.

[122] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh and R. Riley, “Iso-x: A flexible architecture for hardware-managed isolated execution,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[123] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally and L. Torres, “Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines,” in *Transactions on Computational Science IV: Special Issue on Security in Computing*, M. L. Gavrilova, C. J. K. Tan and E. D. Moreno, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 1–22.

[124] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and A. Martinez, “A parallelized way to provide data encryption and integrity checking on a processor-memory bus,” in *Proceedings of the 43rd annual Design Automation Conference*, 2006.

[125] G. Dessouky, A.-R. Sadeghi and E. Stapf, “Enclave Computing on RISC-V: A Brighter Future for Security?”.

[126] D. Champagne and R. B. Lee, “Scalable architectural support for trusted software,” in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010.

[127] B. Ali and A. I. Awad, “Cyber and Physical Security Vulnerability Assessment for IoT-Based Smart Homes,” *Sensors,* vol. 18, 2018.

[128] B. S. X-Lab, “PC security facing another “heavy hammer”, Baidu Security discovers a new Rowhammer attack,” 2019. [Online]. Available: https://medium.com/baiduxlab/pc-security-facing-another-heavy-hammer-baidu-security-discovers-a-new-rowhammer-attack-be3dce8d1e92. [Accessed 12 January 2022].

[129] P. L. Dordal, “An Introduction to Computer Networks, Security,” 2021. [Online]. Available: http://intronetworks.cs.luc.edu/current/html/security.html. [Accessed 12 January 2022].

[130] B. Aveling, “Swiss cheese model,” 2020. [Online]. Available: https://commons.wikimedia.org/wiki/File:Swiss_cheese_model.svg. [Accessed 12 January 2022].

[131] A. One, “Smashing the stack for fun and profit,” *Phrack magazine,* vol. 7, p. 14–16, 1996.

[132] O. Mutlu and J. S. Kim, “Rowhammer: A retrospective,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 39, p. 1555–1571, 2019.

[133] S. B. O. D. H. M. P. R. F. R. A. S. Roberto Avanzi, “Protecting Memory Contents on ARM Cores,” New York, 2020.

[134] A. Costin and J. Zaddach, “Iot malware: Comprehensive survey, analysis framework and case studies,” *BlackHat USA,* 2018.

[135] C. Kolias, G. Kambourakis, A. Stavrou and J. Voas, “DDoS in the IoT: Mirai and other botnets,” *Computer,* vol. 50, p. 80–84, 2017.

[136] D. Goodin, “When coffee makers are demanding a ransom, you know IoT is screwed,” Ars Technica, 26 September 2020. [Online]. Available: https://arstechnica.com/information-technology/2020/09/how-a-hacker-turned-a-250-coffee-maker-into-ransom-machine/. [Accessed 20 February 2022].

[137] M. Morbitzer, M. Huber, J. Horsch and S. Wessel, “Severed: Subverting amd's virtual machine encryption,” in *Proceedings of the 11th European Workshop on Systems Security*, 2018.

[138] W. Shi, H.-H. S. Lee, M. Ghosh and C. Lu, “Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems,” in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 2004.