An In-Depth Study and Implementation of Protection Mechanisms For Embedded Devices Running Multiple Applications

Abderrahmane Sensaoui

To cite this version:

Abderrahmane Sensaoui. An In-Depth Study and Implementation of Protection Mechanisms For Embedded Devices Running Multiple Applications. Embedded Systems. Université Grenoble Alpes [2020-..], 2020. English. NNT : 2020GRALM002 . tel-02923713

HAL Id: tel-02923713
https://theses.hal.science/tel-02923713
Submitted on 27 Aug 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Abderrahmane Sensaoui

Thèse dirigée par **Oum El Kheir AKTOUF**, Maître de Conférences, Communauté Université Grenoble Alpes
et codirigée par **David HELY**, Maitre de Conférence, Communauté Université Grenoble Alpes

Préparée au sein du **Laboratoire de conception et d'intégration des systèmes**
dans l'École Doctorale **Mathématiques, Sciences et technologies de l'information, Informatique**

Etude et implémentation de mécanismes de protection d'exécution d'applications embarquées

An In-Depth Study and Implementation of Protection Mechanisms For Embedded Devices Running Multiple Applications

Thèse soutenue publiquement le **24 Janvier 2020,**
devant le jury composé de :

Madame MARIE-LAURE POTET
PROFESSEUR, GRENOBLE INP, Présidente du jury
Madame OUM-EL-KHEIR AKTOUF
MAITRE DE CONFERENCES HDR, GRENOBLE INP, Directrice de thèse
Monsieur DAVID HELY
MAITRE DE CONFERENCES HDR, GRENOBLE INP, Co-directeur de thèse
Monsieur BRUNO ROUZEYRE
PROFESSEUR, UNIVERSITE DE MONTPELLIER, Rapporteur
Monsieur DANIEL HAGIMONT
PROFESSEUR, INP TOULOUSE - ENSEEIHT, Rapporteur
Monsieur LILIAN BOSSUET
PROFESSEUR, UNIVERSITE JEAN MONNET - SAINT-ETIENNE, Examinateur

An In-Depth Study and Implementation of Protection Mechanisms For Embedded Devices Running Multiple Applications

by

Abderrahmane Sensaoui

Submitted to the Université Grenoble Alpes
on January 24, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Looking at the speed by which embedded systems technologies are advancing, there is no surprise the attacks' number is rising. Many applications are written quickly in a low-level language to keep up with industry pace, and they contain a variety of bugs. Bugs can be used to break into a device and to run malicious code. Reviewing code becomes more and more complex and costly due to its size. Another factor complicating code review is the use of on-the-shelf libraries. Even a detailed code review does not guarantee a bug-free application.

This thesis presents an architecture to run securely untrusted applications on the same platform. We assume that the applications contain exploitable bugs, even the operating system can be exploited. We also assume that attackers can take control of In/Out hardware components (e.g., Direct Memory Access (DMA)). The device is trusted when the architecture guarantees that attackers cannot compromise the whole device and access sensitive code and data. Even when an application is compromised, our architecture guarantees a strong separation of multiple components: hardware and software. It ensures the authenticity and integrity of embedded applications and can verify their state before any sensitive operation. The architecture guarantees, for local and remote parties, that the device is running properly, and protect against software attacks.

First, we study multiple attack vector and isolation and attestation architectures. We present multiple software attack vectors, and we define the security features and properties that these architectures need to ensure. We provide a detailed description of fifteen existing architectures in both academia and industry, and we compare their features. Then, we provide an in-depth study of five lightweight architectures where we give a comparison of performance, size, and how they behave against software-based attacks. From these studies, we draw our security objectives for lightweight devices: *multi-layer isolation*, *attestation*, *upgradability*, *confidentiality*, *small size with a negligible run-time overhead* and *ease-of-use*.

Then, we design hybrid isolation and attestation architecture for lightweight devices. The so-called Toubkal offers multi-layered isolation; the system is composed of three layers of isolation. The first one is at the hardware level to separate In/Out components from each other. The second one is at the security monitor level; our study shows that there is a strong need to create a real separation between the security monitor and all the rest. Finally, the third layer is at the application level.

However, isolation itself is not sufficient. Devices still need to ensure that the running application behaves as it was intended. For this reason, Toubkal provides attestation to be able to check the state of a device at any-time. It guarantees that a software component or data were not compromised.

Finally, we prove the correctness of the security properties that Toubkal provides. We modeled Toubkal as a finite state machine and used computer-aided formal verification to prove the security properties. Then, we evaluated Toubkal's overhead. The results show that Toubkal overhead is small and fit for lightweight devices.

Thesis Supervisor: Oum-El-Kheir Aktouf
Title: Associate Professor

Thesis Supervisor: David Hely
Title: Associate Professor

Résumé en Français

En considérant la vitesse avec laquelle la technologie des systèmes embarqués progresse, il n'est pas étonnant que le nombre des attaques des systèmes soit en nette augmentation. De nombreuses applications sont développées rapidement et sont écrites avec un langage bas niveau pour suivre le rythme avec lequel progresse l'industrie des systèmes embarqués. Souvent, ces applications contiennent beaucoup de bugs. Certains bugs peuvent être exploités pour pénétrer un système et exécuter un code malveillant. Aujourd'hui, la revue de code peut s'avérer très coûteuse vu la taille des codes développés. En outre, une revue détaillée de code ne garantit pas un système infaillible.

Cette thèse présente une architecture permettant l'exécution de plusieurs applications sécurisées et non sécurisées sur une même plate-forme « légère ». Notre architecture doit garantir que même s'il y a une application compromise, les attaquants ne peuvent pas compromettre la totalité du système et/ou récupérer les données des autres applications. Elle doit garantir une forte séparation entre tous les périphériques et les applications présents sur la plate-forme. Finalement, elle doit aussi être capable de vérifier l'état de n'importe quel bout de code. Pour pouvoir garantir ces points, nous utiliserons des techniques d'isolation et d'attestation.

Dans un premier temps, nous avons étudié plusieurs architectures d'isolation et d'attestation décrites dans la littérature et utilisés par l'industrie. L'étude a montré qu'il existe une grande variété d'architectures intéressantes offrant différents niveaux de protection et visant différents systèmes. Les systèmes avec une grande capacité de calcul proposent un bon niveau de protection. Par contre, les systèmes « légers », qui ont des ressources très limitées et doivent répondre aux contraintes temporelles, échouent dans au moins un des critères suivants : l'isolation, les performances, le coût, ou bien la flexibilité.

À l'issue de cette étude, nous avons conçu Toubkal. Une solution hybride (Co-design logiciel et matériel) pour offrir une architecture d'isolation et d'attestation modulaire qui permet d'établir une isolation sur plusieurs niveaux, de détecter la présence d'un logiciel malveillant ou une donnée malveillante avec des performances acceptables et un coût réduit.

Toubkal est principalement composé de trois modules ; deux matériels et un logiciel. Le premier module, appelé Master Memory Protection, permet de créer un premier niveau d'isolation pour contrôler les accès mémoire des périphériques. Le deuxième module, appelé Execution Aware Protection, permet de renforcer la protection d'un logiciel critique, y compris le système d'exploitation.

Ces deux niveaux d'isolation permettent de réduire la surface d'attaque.

L'isolation toute seule ne suffit pas pour garantir que les applications fonctionnent comme il le faut. En fait, l'attaquant peut toujours modifier le comportement d'une application faillible. Pour cela, Toubkal propose un root immuable qui permet d'attester l'intégrité des autres applications.

Pour valider le design de Toubkal, nous avons défini des propriétés de sécurité que nous avons prouvé avec la vérification formelle. Nous avons aussi évalué la taille de Toubkal. Les résultats montrent que le coût de Toubkal est acceptable pour un système dit « léger ».

Finalement, nous avons conclu cette thèse avec une discussion des limitations de Toubkal et les perspectives pour améliorer le design et offrir plus de protection, comme par exemple le chiffrement du code à coût caché.

Acknowledgments

Too many times, we skimp on expressing or showing our gratitude to others because things become routines or because we think the person already knows how thankful we are towards them. Therefore, by thereafter, I wish to express all my gratitude for all the amazing people involved, directly or indirectly, in this Ph.D.

I would like to express my deep and sincere gratitude to my doctoral advisers Oum-El-Kheir Aktouf and David Hély for accepting me to work with them. They were very considerate and gave me valuable lessons and tips to carry out this Ph.D. I am very thankful for the time they allocated me for various project meetings and their continuous support, especially during rough times. Without their help and guidance, I would not be able to finish this Ph.D. in a lesser time than it was supposed.

I would like to express my gratitude to Yann Loisel and Frank Lhermet for offering me the chance to start this Ph.D., for all the trust I was given throughout these years and also for the time for various meetings to discuss and review this project. I would like to thank Stephane Di Vito for his support and his time to review and discuss my work. I would also like to thank Vincent Albanese, and all the team in La Ciotat for their time and support to finish this project.

I would also like to thanks the jury members for the time and effort spent in reviewing my work.

Besides these awesome people I had the chance to work with, I would like to express my special thanks to my family and friends who supported and encouraged me during difficult times.

I would like to thank Maxim Integrated for supporting this work, and the LCIS laboratory and the doctoral school for supporting my work and conferences' trips when things got complicated at the company.

Acronyms

CAM	Content-Addressable Memory
CPU	Central Processing Unit
CWE	Common Weakness Enumeration
DMA	Direct Memory Access
DRoT	Dynamic Root of Trust
EAP	Execution Aware Protection
FSM	Finite State Machine
GE	Gates Equivalent
IO	In/Out
IP	Intellectual Property
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
LOC	Line Of Code
LTL	Linear Temporal Logic
MMP	Master Memory Protection
MMU	Memory Management Unit
MPU	Memory Protection Unit
NVD	National Vulnerability Database
OS	Operating System
OTP	One Time Programmable
PC	Program Counter
PMP	Physical Memory Protection
RLE	Run-Length Encoding
ROM	Read Only Memory
ROP	Return Oriented Programming
TCB	Trusted Computing Base
TLB	Translation Look-aside Buffer
TOCTTOU	Time-of-check-to-time-of-use

Contents

1 Introduction 21

1.1 Memory Isolation 22

1.2 Software Attestation 23

1.3 Thesis Contributions 24

1.4 Publications 26

1.5 Outline of this Thesis 27

2 Lightweight Devices Security Challenges 29

2.1 Lightweight Devices 29

2.2 Security Threats 31

2.2.1 Software based attacks 31

2.2.2 In/Out Peripherals based attacks 37

2.2.3 Threat Model 38

2.3 Goals to secure a lightweight device 39

2.3.1 Properties for an effective isolation and attestation architecture 40

2.3.2 Use Cases 41

2.4 Summary 42

3 Isolation and Attestation Architectures 45

3.1 Background 45

3.2 Security Properties 48

3.3 Architectures 49

3.4 Comparison 67

3.5 Summary 71

4 **In-depth MPU-based Architectures Comparison** **73**

4.1 Comparison Criteria 73

4.2 Security Protection 75

4.3 Trusted Computing Base 78

4.4 Performance and Memory Consumption 79

4.4.1 Performances 79

4.4.2 Memory consumption 82

4.5 Discussion 83

4.6 Summary 86

5 **Toubkal: Hardware-based Isolation and Attestation Architecture** **87**

5.1 Toubkal Overview 88

5.1.1 Toubkal Architecture 89

5.1.2 Modularity 91

5.2 Toubkal Design in Detail 92

5.2.1 The Master Memory Protection 92

5.2.2 The Execution Aware Protection 95

5.2.3 The Immutable root 101

5.3 Summary 104

6 **Toubkal Security Analysis and Evaluation** **105**

6.1 Toubkal Security Analysis and Validation 105

6.1.1 Security Properties 106

6.1.2 Execution Isolation Computer-Aided Verification 107

6.1.3 Asset Protection and Confidentiality Computer-Aided Verification 110

6.1.4 Software Attestation Verification 112

6.2 Toubkal Evaluation 112

6.2.1 The Master Memory Protection Footprint 112

6.2.2 The Execution Aware Protection Footprint 116

6.2.3 The root Footprint and performance 116

6.3 Summary 116

7 Toubkal design discussion and perspectives for future works **119**

7.1 Discussion 119

7.1.1 Comparison to Existing Architectures 121

7.2 Future Work 123

7.2.1 Enforced Access Control 123

7.2.2 Enforced Attestation 125

7.2.3 Code Confidentiality 126

7.3 Summary 126

8 Conclusion **129**

List of Figures

2-1 Simple buffer overflow example 33

2-2 Call stack operation 34

2-3 Stack-based buffer overflow: On the left, a snippet of the stack before executing the `strcpy()`. And on the right, a snippet of the stack after executing the `strcpy()`. The buffer overflow leads to a change of the return address. Therefore, the path has been hijacked and when the subroutine finishes its execution, instead of returning to the calling routine, it will jump to the address 0x100014ab. 35

2-4 A visual of how a single contiguous memory space is segmented between different Masters. 41

3-1 MPU configuration: The MPU in the ARMv7-M architectures requires that the memory region size must be aligned to a start address. This address is a multiple of the region size, and the region size must be a power of two. In Example 2, we can see the hole between the second and the third region. 46

3-2 MoMP architecture: It is mainly composed of a Translation Look-aside Buffer (TLB), sidecar registers, and a reserved memory area, called Permission Table, to store all memory regions configuration. When the processor tries to access a memory region, it checks within the sidecar register if there is a match or not. If there is no match or the permissions are not valid, MoMP tries to reload the sidecar from the TLB. If the TLB does not have the permissions configuration, the MoMP tries to look them up from the reserved memory area storing all configurations. 47

3-3 SMART architecture: SMART is composed of the memory backbone, and the ROM to store immutable code and a cryptographic key. The memory back bone controls all memory accesses, its primary goal is to block any attempt to retrieve the cryptographic key or run the ROM code in an unexpected way. 50

3-4 Sancus model: The infrastructure provider (IP) provides multiple nodes (in grey). Each node can contain multiple protected software modules (SM). Sancus uses software providers (SP) to load SMs within the nodes. Each SP and node can derive a cryptographic key to load protected SMs. An SM is mainly composed of an entry point and code and data memory regions. The SMs have to be called from their entry point. 52

3-5 TyTAN architecture 54

3-6 TrustLite architecture 55

3-7 uVisor architecture 57

3-8 TockOS architecture 59

3-9 TrustZone-M architecture 61

3-10 Aces architecture 65

4-1 uVisor and RTOS have same privileges: RTOS is non secure, and contains many buffers, but, it has the same privileges as uVisor, which can be critical to the system protection. 77

5-1 Effective isolation and attestation architecture overview 88

5-2 Major Components of Toubkal: It is a hardware/software co-design composed of three main components; the MMP which is the hardware module control memory access of the different peripherals, like CPUs, DMA, etc. The EAP which is the hardware module to protect and create a strong separation of the security monitor from all the rest. Finally, the security monitor which is the only software responsible of enforcing system security. The PMP is optional and can be used to create multiple zones for the user mode. 89

5-3 Toubkal's architecture 90

5-4 Major components of hardware block of the MMP: The uCode module contains information to recognize the security monitor which is responsible of configuring the MMP. This module is linked to the MLB. The MLB is a set of registers to store regions configurations. The abstraction layer is customizable and aims to facilitate the integration of the MMP in different systems and at different spots. 93

5-5 MLB slots: The figure presents the different ways to store configurations in the MLB for a 32 bit address space. N represents the number of masters. The first method, called UMLB (U for Unique) stores one configuration per Master per slot, while the second method, called SMLB (S for Shared), stores one configuration per slot but for multiple masters. In the SMLB, there is no need for a valid bit as each Master has a corresponding bit in the first column from the left. The value 0 means the configuration is not valid for that master and vice versa. The third column from the left refers to size in case of coarse-grained granularity, and to end address in case of fine-grained one. 93

5-6 A fine grained match is more complex than a coarse grained match. The second one compares the address with the base address and the limit address, while the first one compares the logical combination of the address and the mask corresponding to the size of the region with the base address. 95

5-7 Registers to configure the protected text sections: Regions cannot overlap. Entry points to the protected region must be inside the start address and End_Entry. Entry points must be as few as possible to reduce attack surface. 96

5-8 The EAP is mainly composed of two components. The *state transitions* component is responsible of checking if the transition from oldState to newState is authorized. Then, the second component, *memory control*, is responsible of checking the memory access data if it is authorized from that state. 99

5-9 States Transitions: Black arrows represent authorized transitions. Any other attempt will lead to a transition to sKill state as illustrated in red arrows. When the EAP reaches the state sKill, it resets the system. 101

5-10 This figure presents the keys used to compute the hash to verify the integrity of each application. 102

5-11 Software Attestation consists of retrieving memory region information from the header, computing memory region hash, and comparing the computed hash with the provided one. 102

6-1 State sequence correctness around the state sRomIn 108
6-2 State sequence correctness starting from the state sMonitorIn 109
6-3 Code and keys confidentiality 110
6-4 System Architecture Overview. 113

7-1 context-switche main steps: 1) The _Caller calls the gateway of the _Callee. The gateway in escalates the privilege, saves the _Caller context and loads the _Callee context. 2) The gateway de-privileges the execution and jumps to the _Callee entry. The _Callee does its job. Then, it calls the gateway out. 3) The gateway out escalates the privilege again, loads the stored context of _Callee. 4) The gateway out de-privileges the execution and jumps to the return address in the _Caller. . . . 124
7-2 Remote attestation scheme. 125

List of Tables

3.1 Summary of the studied isolation and attestation architectures. 70

4.1 Summary of Security protection studied in this chapter: 78
4.2 Performance of creating a process (in clock cycles). 80
4.3 Performance of configuring the MPU (in clock cycles). 80
4.4 Performance of peripheral registers writing (in clock cycles). 81
4.5 Performance of Interrupts switch in and switch out (in clock cycles). 81
4.6 Performance of context switching (in clock cycles). 81
4.7 Performance of uVisor MPU recovery mechanism (in clock cycles). 82
4.8 FLASH memory consumption (in kB). 82
4.9 Initial RAM memory consumption (in kB). 83
4.10 Summary of the evaluated MPU-based isolation architectures. 85

5.1 Notations used in designing and verifying the state machine of EAP 97
5.2 Memory access control for all rings 97

6.1 MMP area for a dynamic policy (in kGE) 114
6.2 MMP area for a dynamic policy (in kGE) 114
6.3 MMP area for different policies (in kGE) 115
6.4 MMP area for semi-static policy (in kGE) 115

7.1 Summary of Toubkal's comparison to the studied isolation and attestation architectures. 122

Chapter 1

Introduction

In 2018, ReSwitched, a team of security researchers, published a report on a vulnerability allowing malicious software to execute in privileged mode. The report shows that malicious code can have access to the whole memory and can compromise the whole root-of-trust for each processor. An attacker could take advantage of a software vulnerability to copy the contents of their controlled buffer overflow in the execution stack and run their contents in high privileges. This is one example of many of how systems that we use every day can be compromised.

There is a lot of effort and research to analyse software components and detect vulnerabilities. However, it is challenging to guarantee large software that they are bug-free. Hence, all software components must be treated as equally untrusted pieces of software and must be protected from security threats. Even software that is running in privileged mode has to be considered untrusted and in a compromised state.

Nowadays, the use of lightweight devices is increasing in the Internet of Things, health care, and automotive. Most of used devices contain sensitive information like cryptographic keys, intellectual property, and private data. To build a resilient device, we need enough layers of protection: layers to protect the hardware, the software, and the internal and external communications between peripherals and devices.

In this regard, isolation and attestation can play a vital role. On the one hand, isolation is a technique that aims at dividing the software into multiple separated environments. Here, each environment is considered as memory resources, and each one cannot access other resources. The primary purpose of isolation is to limit the consequences in case an exploited bug exists in a specific software component. For example, attackers can leak or overwrite sensitive data, or they can inject malicious code to take control over the whole device. On the other hand, attestation is a technique that aims at verifying the state of the device itself and other devices. The main goal of attestation is to detect malicious software components and data and prevent them from breaking the device security.

1.1 Memory Isolation

Today's devices rely on many solutions to protect memory regions. Early research relied on hardware components such as the Memory Protection Unit (MPU) to create separate contexts and isolate different applications. These components run the application in non-privileged mode and limit the application memory access to some regions. There are different schools of MPU usage, some are manual [9, 18, 32, 48, 102], others are more automatic [22, 23, 47], although the aim is the same.

However, these architectures have safety and security limitations. First, the privileged modes have access to the whole memory mapping, which can be critical to the system security and dependability. Many researchers [2, 9, 99] have shown flaws in Operating Systems (OS) and how an attacker can escalate the privileges, and so, they can access all secrets stored in memories, change the MPU configuration... Second, they only offer controlled memory accesses to the Control Processing Unit (CPU). Unfortunately, the CPU is not the only hardware component connected to memories. Therefore, an additional layer of isolation is needed.

Interesting solutions [10, 33–35, 64, 65] have been proposed by some academic papers and patents to add a particular Memory Management Unit (MMU) for peripherals such as the Direct Memory Access (DMA) peripheral. The so-called In Out MMUs (IOMMU) or System MMU (SMMU) offer solutions similar to the traditional MMU. They add address translation and permissions for IO components.

However, these solutions are subject to a significant limitation in this thesis context. They are destined for high-performance computing systems and require lots of cells' area and memories as they use page tables and caches. Moreover, the IOMMUs and SMMUs are interfaced with one IO component (such as DMA). Therefore, if there are two IO components of which accesses must be controlled, the device will need two IOMMUs or SMMUs.

Another limitation they suffer from in this context is the latency introduced into the devices. According to [16], they impose an extra performance penalty. This penalty is caused by the various mapping and un-mapping calls to create translation entries in the IO component address space.

Contrary to high-performance systems, micro-controllers have a strong power consumption and size constraints, and most of their applications are time-critical. Using any above cited solution will add high overheads, power consumption, hardware area, and run-time.

Others [26, 27, 35, 53] offer a DMA transfer filter where they define a safe region in the memory controller for the DMA. The DMA can only access the defined region. However, this solution does not offer a real and complete separation. It limits the DMA access to a contiguous memory space.

While many of the lightweight solutions cited above need a software to manage them, a few [18, 48] have investigated this piece of software attestation for its integrity and authenticity.

1.2 Software Attestation

Verifying the state of software and data is an essential task in many fields. For example, medical devices are being increasingly used to improve monitoring and range aid. An attacker can cause considerable damage to patients by changing the software running on the device with a malicious one.

One technique to detect and disable a malicious code is attestation. Today's attestation techniques can fall into 3 categories; hardware only [24, 39, 67, 74], software only [82, 83], or hardware/-software [18, 30, 68, 97] based techniques. The first ones rely on dedicated hardware to perform attestation during boot time (static attestation) or during run-time (dynamic attestation). [24, 39] offer good static solutions, but they are more suited for high-performance devices and can be very expensive for micro-controllers.

The software-based attestation techniques are mostly time-based. They rely on assumptions like the exact time of specific operations and silent adversary (it means, during an attestation, only the prover is communicating with the verifier) which are difficult to achieve in practice and may be unrealistic for many applications.

The hardware/software-based methods combine software and a hardware block to offer attestation. This category is more suitable for micro-controllers as the other categories introduce either high cost or high-performance overheads or both.

Another criteria to consider is if these techniques provide a static attestation or a dynamic one. Static attestation is limited to boot-time comparing to dynamic attestation, so there are no guarantees about the state of the software at a given time. In contrast, dynamic attestation can verify and detect attacks in the software, like Return Oriented Programming (ROP) based attacks. Attackers use ROP to execute arbitrary code by, for example, changing the return address on the stack. While memory isolation can reduce ROP attacks [22], dynamic attestation can verify the state of the stack to make sure the return address was not overwritten. Dynamic attestation can also be beneficial to verify code loaded dynamically.

Concerning hybrid dynamic RoT, the cited techniques have their limitations. For example, SMART [30] is a lightweight solution that offers Dynamic Root of Trust (DRoT) for low-end devices. It uses custom hardware design to control access to a Read-Only Memory (ROM) where the attesting software and the key used for attestation are stored. An issue is that they assume that some IO peripherals can be disabled during SMART execution. But what about the rest of the time? They can read the ROM and read the key. Another issue is that we cannot change the key as it is burned during fabrication.

Other techniques, such as [18], offer DRoT based on the Executive-Aware MPU (EA-MPU). The issue here is that the trusted software responsible for DRoT lives in the Flash and is not attested nor protected from overwriting during boot time.

1.3 Thesis Contributions

One challenge of this thesis is learning from experience to implement an isolation and attestation architecture ready for sophisticated lightweight devices with as few as possible resources. This thesis contributions can be divided into two primary parts. In the first part of this work, we studied lightweight devices threats and isolation and attestation architectures in details. At first, we studied both lightweight devices and high-computing devices. Then, we focused on lightweight devices to draw the needs for a secure architecture. In the second part, we designed a new security architecture for lightweight devices. This architecture is called Toubkal* and its main goal is to offer a multi-layer isolation and protection.

---

* Toubkal is the name of a mountain in Morocco. The reason behind choosing a mountain name is because there are similitude between isolation and attestation architectures and mountains. Depending on the characteristics and climbers tools, they can more or less reach the peak. Same here, depending on the architecture features and attackers abilities, they can more or less break the device security

In detail, we can summarize the first part of our contributions into:

- We studied different security threats of lightweight devices to understand the risks they may encounter and draw protection goals.

- We performed a detailed study of fifteen architectures, and a discussion of their main contributions and limitations

- We conducted an in-depth study of five lightweight architectures. The goal, here, is to see how each architecture impacts on performances and resources use, and how they protect from the security threats presented before.

Based on this detailed study, we design Toubkal, an isolation and attestation architecture. Toubkal consists of designing a hardware software architecture targeting RISC-V Instruction Set Architecture (ISA)x that is composed of two hardware components that are responsible for controlling memory accesses of the core and any other peripheral connected the memory (e.g. DMA); the Master Memory Protection and the Execution Aware Protection, and a software component called the security monitor which is responsible for configuring these hardware components and establishing trust within the device. Therefore, Our contributions in the second part of our work can fall into designing and developing three components:

- We designed the hardware module called Master Memory Protection (MMP), which controls memory accesses of different peripherals connected to memories.

- We designed the hardware module called Execution Aware Protection (EAP), which its primary goal is to offer a strong and flexible isolation and protection of the security monitor. It controls all the jumps into the security monitor and ensures it was called from the defined entry point. it also protects its sensitive information.

- We designed and developed an immutable software, part of the security monitor, which is the only fully trusted software. This component is responsible of configuring the hardware modules and establishing trust in the rest of the software, even parts of the security monitor itself.

1.4 Publications

This thesis is based on works and results from existing publications presented at conferences, workshops, and journals:

Journals

[77] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, David Hely, and Stephane Di Vito. An in-depth study of mpu-based isolation techniques. *Journal of Hardware and Systems Security*, Nov 2019.

Conferences

[81] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Toubkal: A Flexible and Efficient Hardware Isolation Module for Secure Lightweight Devices. In *2019 15th European Dependable Computing Conference (EDCC)*, Naples, Italy, September 2019.

Workshops

[78] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Hardware-based Isolation and Attestation Architecture for a RISC-V Core. In *SiFive's Technical Symposium*, Grenoble, France, May 2019.

[76] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, and David Hely. Shcot: Secure (and verified) Hybrid Chain of Trust to protect from malicious software in lightweight devices. In *The 1st Annual International Workshop on Software Hardware Interaction Faults, co-located with ISSRE 2019*, Berlin, Germany, October 2019.

Posters

[79] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Poster: Hardware-based Isolation and Attestation Architecture for a RISC-V Core. In *2019 CySep and EuroS&P*, Stockholm, Sweden, June 2019.

[80] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Poster: Hardware-based Isolation and Attestation Architecture for a RISC-V Core. In *2019 15th European Dependable*

Computing Conference (EDCC), Naples, Italy, September 2019.

1.5 Outline of this Thesis

This thesis is structured as follows. Chapter 2 defines the context, presents the security threat and the objectives of this work. Then, Chapter 3 presents a detailed description of fifteen isolation and attestation architectures and discusses their contributions and limitations. Chapter 4 studies in-depth five of the fifteen architectures, which target lightweight devices. The study includes the analysis and comparison of the security features they offer, performances, and resource consumption. Chapter 5 presents an overview of the design of our isolation and attestation architecture called Toubkal. The architecture is composed of three components; the first component is responsible for creating the first layer of isolation targeting IO peripherals. The second component is responsible for protecting critical software. And, the third component is the immutable software responsible for checking the integrity and authenticity of the software. Then, chapter 6 analyses and evaluates the proposed architecture. Chapter 7 discusses our architecture and compares it to the existing ones. Then, it discusses the perspective of future works. Finally, chapter 8 concludes this thesis.

Chapter 2

Lightweight Devices Security Challenges

The main objective of this chapter is to define clearly the context of this work. The type of systems we target, the issues they encounter and those we want to tackle, and the boundaries of this context. Hence, this chapter starts with an overview of the studied devices. Then, it presents different widespread security and safety weaknesses. Finally, we present our goals to secure these lightweight devices. The main idea behind this work is, instead of thinking about each specific vulnerability and how to secure it, to think systematically on their impacts, and the architectural changes we need to improve the whole device security and limit the consequences of attacks.

2.1 Lightweight Devices

This section presents the type of targeted devices. The so-called lightweight devices are known for having a flat memory model. The flat memory model is a memory model where the address space is a single and continuous page. These devices do not have any Memory Management Unit (MMU) as the memory managing is not sophisticated, and the devices are resource-constrained (cells area, computational capabilities, memories, energy) and time-critical. Therefore, the whole memory space is accessible for In/Out hardware units such as the Central Processing Unit (CPU), the Direct Memory Access (DMA) and the likes.

These devices have always been prone to multiple attacks. Because of their resource limitation, security is, most of the time, neglected, and they cannot benefit from existing solutions within high-end and sophisticated devices. Some solutions, such as Control-Flow Integrity and Data-Flow Integrity [29], have been proven effective, but they are not deployable by industry because they require CPU changes [4, 28, 73, 98]. However, changing the hardware may require licenses which can be very costly.

Challenges in lightweight devices

Lightweight devices' security is hard to establish. There are multiple factors and challenges that designers and developers face while designing and developing their products. Among such challenges, we have: the cost of the device, securing sensitive data, making sure that the code runs the way it is intended, and how to establish trust of the device itself and while communicating with other devices.

Cost: Industrials produce lightweight devices on a large scale. Therefore, they try to reduce the cost as much as possible, and sometimes at the expense of security. The cost of the device is correlated to its resources; memory and silicium resources. An increase of one resource might result in a huge final cost at a large scale.

No Separation: Lightweight devices have a flat memory model. In this memory model, with a compiler and a linker, a lot of code blocks and applications are combined into one memory image. This way is convenient to attackers, as they can access to any memory address they want, and then leak secrets. Because there is no separation, a single flaw in one application or library can propagate quickly to the whole device.

Hard Verification: The attack surface is big enough to make verification very hard and costly. The lack of separation and secure interfaces between the different components and applications require an exhaustive verification that, in most cases, is impossible to carry out.

Trust: Today, devices are increasingly connected and communicate with external components. This adds other entry points for attackers. For example, a malicious component can send over a communication protocol a malicious data to a device and change its normal behaviour, or break it. How can we establish trust between devices and components to make sure they are not compromised?

2.2 Security Threats

This section presents in detail the different security threats we took into account in this thesis. We can divide the studied threats into two main categories, Software-based threats and IO Peripherals based threats. Then, we define this work threat model.

2.2.1 Software based attacks

We studied several vulnerabilities and exploits from different papers [52, 70, 91, 99], and from the National Vulnerability Database (NVD) [69]. The NVD groups vulnerabilities from the Common Weakness Enumeration (CWE) specifications. It is a list of common security weaknesses that serve as a reference point for vulnerabilities identification, mitigation, and prevention effort. We also analysed open issues and bugs in some architectures' GitHub repositories [9, 102] to identify other weaknesses.

For risk evaluation, vulnerabilities must be identified, and for better identification, one can classify vulnerabilities. Our study of vulnerabilities and exploits shows that attackers can target different areas to compromise a system, we call these areas *security hotspots*. A *security hotspot* describes the sensitive areas where security is more critical than in other areas. This helps us to define what part of a device is targeted by a certain attack. It clarifies and helps to identify the best opportunities to compromise a device. Main security hotspots are *Authentication, Memory, Cryptography, Logic Errors, Synchronization and Timing, and Validation*. We chose to focus on the *Memory hotspot* because it is the most exploitable hotspot and has an important impact on lightweight devices security. We present some issues and vulnerabilities in this hotspot. These can be divided into two main categories: *Temporal* and *Spacial* errors.

Temporal errors

This category concerns attacks that take advantage of allocating, freeing and deleting memory chunks. For example, in this category, there are *use-after-free* and *double free* vulnerabilities. If exploited, they can lead to information leakage or to control flow hijacking or even to crash the program.

Use-after-free: A use-after-free vulnerability is when a program keeps using a pointer to a memory chunk that is not allocated anymore, and possibly re-allocated by another part of the program. The exploitation of this vulnerability can go from no effect to the execution of an arbitrary code.

Double free: A double free leads to an undefined behaviour. This vulnerability happens when the same memory block is freed twice. Double frees can occur in different cases. For example, when two or more pointers point to the same memory block, and begin cleaning using `free()`. The developer, if not careful, might free the same pointer many times. This might cause for other existing memory spaces to get corrupted or to fail future allocations.

Spacial errors

This category focuses on spatial errors. Attackers can exploit spatial errors to execute code, to read/write the stack, or to halt the system. This can lead to unwanted behaviour, and could extract sensitive data or change the defined program flow. In this category, vulnerabilities like *buffer overflows*, *stack overflows*, *heap overflows*, *format string*, *truncation*, and *signed convention* can be found.

Buffer overflow: A buffer is a memory area fixed to contain data. A buffer overflow occurs when the data written into a buffer overruns the buffer boundaries. And because of the contiguous memory space in lightweight devices, if the boundaries are not checked properly before writing or reading a buffer, it overwrites neighbouring memory locations.

Most software developers have no security background; they can create unsafe code that leads to a buffer overflow. For example, the following code shows an example of an unsafe buffer copy that causes a buffer overflow:

```
char buffer[3] = "AAA";
char x = 'a';

strcpy(buffer, (char *)"TRUSTED");
```

Figure 2-1 illustrates a bit of the memory after we finish copying data into the buffer. The value of the variable x is overwritten. The results of exploiting a buffer overflow vary depending on the location of the overflowed buffer within the memory.

Stack-based buffer overflow: Stack-based buffer overflow is a buffer overflow where the program allocates the overflowed buffer on the stack. Attackers can exploit stack-based buffer overflows to manipulate the control flow of a program. When the system calls a function, a stack frame

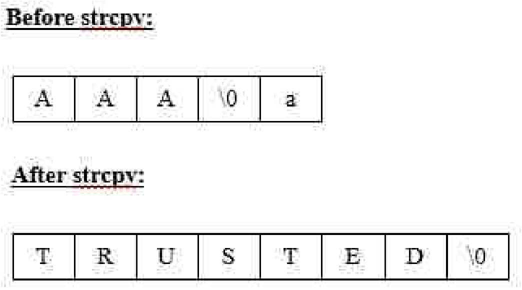


Figure 2-1: Simple buffer overflow example

saves information related to this function. Then the stack frame contains information like the function parameters, local variables, and the return address. When a buffer stored in the stack is overwritten, the attacker can change data within the stack frame, especially the address stored within the return address. This allows the attacker to gain control over the execution path of the program. The following function shows an example of how an attacker can change the return address:

```
void function(void)
{
        char buffer1[8];
        char buffer2[16] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
             0x00, 0x00, 0x00, 0x10, 0x00, 0x14, 0xab};

        strcpy(buffer1, buffer2);
}
// some routine executed
function();
// rest of the routine
```

Before explaining what occurs when the function is executed, we explain the stack manipulation in general, when a function is called, the following occurs (Figure 2-2 illustrates a stack frame):

- Push function arguments if there are any.
- Push the return address after the execution of function is finished.
- Allocate space for local variables.

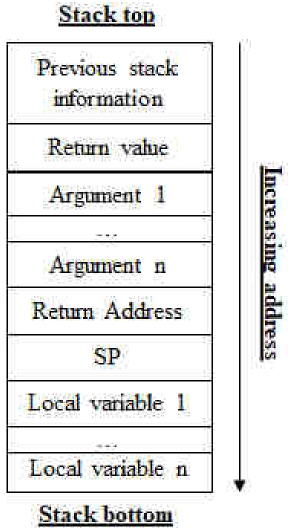

Stack top
Previous stack information
Return value
Argument 1
...
Argument n
Return Address
SP
Local variable 1
...
Local variable n
Stack bottom
Increasing address

Figure 2-2: Call stack operation

The stack grows from higher memory to lower memory (Figure 2-2), and the `strcpy()` starts copying data from a base address towards higher addresses. So, as Figure 2-3 shows, when `buffer2`, which has enough memory space for 16 bytes, is copied into `buffer1` which has a memory area for only 8 bytes, the overflowed 8 bytes will overwrite data on the stack, and the attacker can change the value of the return address, and hijack the execution path.

With a stack buffer overflow, attackers can inject code into the stack and redirect the path into this code. But some OSes provide defenses against code execution on writeable sections. However, many techniques have been developed to overcome these defenses. Return Oriented Programming (ROP) is one of them; the attacker tries to reuse functions and gadgets. It is difficult to stop these attacks because the attacker executes valid existing codes or sequences.

Lower boundary check: Usually, one side of the boundaries is checked, and it's the upper one. What about the other one? The following piece of code presents an example of how not checking both boundaries can lead to exploitation:

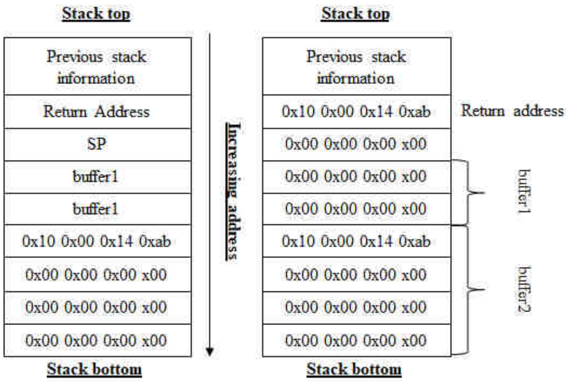

Stack top
Previous stack information
Return Address
SP
buffer1
buffer1
0x10 0x00 0x14 0xab
0x00 0x00 0x00 x00
0x00 0x00 0x00 x00
0x00 0x00 0x00 x00
Stack bottom
Increasing address
Stack top
Previous stack information
0x10 0x00 0x14 0xab
Return address
0x00 0x00 0x00 x00
0x00 0x00 0x00 x00
0x00 0x00 0x00 x00
buffer1
0x10 0x00 0x14 0xab
0x00 0x00 0x00 x00
0x00 0x00 0x00 x00
0x00 0x00 0x00 x00
buffer2
Stack bottom

Figure 2-3: Stack-based buffer overflow: On the left, a snippet of the stack before executing the `strcpy()`. And on the right, a snippet of the stack after executing the `strcpy()`. The buffer overflow leads to a change of the return address. Therefore, the path has been hijacked and when the subroutine finishes its execution, instead of returning to the calling routine, it will jump to the address 0x100014ab.

```
char read_char(char * buff, int buff_length, int index)
{
        if(index < buff_length)
                return buff[index];

        return -1;
}
```

If the attacker somehow controls the value of the parameter `index`, they could load `index` with a negative value and read 8 bits from the memory. The function `read_char` only checks that the given entry `index` is smaller than the given buffer length and does not check the sign of `index` or its lower limit.

Format String: The format string vulnerability occurs when there is a mismatch between the format string parameters (like `%s %x`) and the arguments of the format string function (like, for instance, `print()`). Exploitation of this weakness could lead, for example, to executing a code or reading the stack. The format string function does not have a limitation of entries. For each format string parameters, the format function will need an argument, if there are more format string parameters than arguments, the function will keep fetching data that does not exist in this function call stack. The following example shows how this vulnerability may crash a program:

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s");
```

The printf() will fetch for each `%s` a data from an address within the memory until a string termination character `0x00` is found. There is the possibility that the address from which the printf() will try to read is an illegal address. It results in crashing the system.

An attacker can also read the stack, for example, in the following code, the `printf()` function will retrieve five parameters from the stack and print them on the screen:

```
printf("%x %x %x %x %x");
```

Truncation: Truncation happens during conversion from a larger type to a smaller one. Data is usually lost, and sometimes it can lead to manipulate a branch condition just like the following example shows:

```
void my_function(void)
{
        unsigned short size_temp;
        unsigned char * temp;
        unsigned char buff[512];

        temp = get_username();
        size_temp = strlen(temp);

        if( size_temp > sizeof(buff)) {
                error(0);
        }
        strcpy(buff, temp);
}
```

The variable `size_temp` is an unsigned short, it is included in the range [0, 65535]. If `get_username()` returns a buffer with 65900 characters, `size_temp` will have the value of 364, which is smaller than `sizeof(buff)`. This results in a *buffer overflow*.

Signed Convention: Many Spacial weaknesses are consequences of conversions between signed and unsigned versions of the same type. When for example, a signed integer is converted to an unsigned integer or vice versa, the value can change. The signed integers go from -2 147 483 648 to 2 147 483 647, while the unsigned integers go from 0 to 4 294 967 295. When a conversion occurs, the code checks the most significant bit to decide whether the value will be positive or negative. Consider the following code:

```
void read_data(unsigned int cursor, unsigned char * buffer, unsigned int length);

void my_function(int cursor, int length)
{
        unsigned char buffer[1024];
        // some routine
        if(length > 1024) {
                error(0);
        }
        read_data(cursor, buffer, length);
        // some routine
}
```

In this example, `read_data()` takes `length` as an unsigned integer, `my_function()` takes `length` as a signed integer. If an attacker feeds `my_function()` with a negative value of `length`, the length check can be bypassed. But when `read_data()` is called, a conversion from signed to unsigned occurs, and the value will turn into a large unsigned value.

2.2.2 In/Out Peripherals based attacks

In this category, we focus on attacks mounted using an In/Out (IO) peripheral. The attacker can bypass existing memory protection units to access sensitive memory regions. IO peripherals are all connected to memories with an interconnect bus. Such IO peripherals are the CPU and the Direct Memory Access (DMA) or any other component with direct access to memories.

Currently, most IO attacks abuse the DMA ability to access all memory space directly. In this part, we will focus mostly on DMA attacks with an overview of other IO attacks.

Direct Memory Access

DMA is used in lightweight devices to relieve the CPU from being involved in all memory accesses. The DMA allows multiple peripherals and applications to have access to the whole physical memory space while bypassing the CPU and, therefore, its memory protection mechanisms. For this reason, attackers increased their interest in abusing it to break into a device.

There are multiple ways to mount a DMA attack [58, 75, 95]. For example:

Software vulnerabilities: An attacker can take advantage of a software vulnerability, like buffer overflows, to read from memory a large buffer using a DMA controller. Therefore, they bypass the CPU and its memory protection mechanisms. Attackers can use the DMA to overwrite the security monitor or the OS, then, take full control of the device. In case they cannot control directly the DMA, there is also the possibility to inject a malicious code that will be executed by the CPU, set up the DMA controller, and read/write from memory. Over the years, we have seen multiple concrete examples [3, 45].

Abusing the IO controller: An attacker can carry out non-authorized memory access by abusing the DMA engine interface of an IO controller. The DMA engine is attached to multiple peripherals. Abusing the peripheral controller can lead into using the DMA to access the memory directly.

Malicious IO Peripheral

In this part, we can divide the attacks into two categories:

Malicious external IO Peripheral: Here, the attacks is carried out using an external IO Peripheral. The external Peripheral is interfaced with the memory using a specific DMA controller. By connecting a malicious peripheral, one can read/write directly the physical memory. Therefore, they can leak sensitive data, or rewrite part of the code.

Malicious internal IO Peripheral: This case is frequent in heterogeneous systems. However, it can also happen in lightweight devices. For economic reasons, some hardware manufactures outsource integrated circuit (IC) fabrication or buy ready block designs to integrate. Attackers may exploit these choices to introduce malicious blocks. In this case, the malicious IO peripheral can read/write memory and inject malicious code. A lightweight device produced at a large scale with a malicious IO peripheral will present drastic losses for industrials.

2.2.3 Threat Model

Here, we define our adversarial capabilities. We present what we consider attackers are able to perform to try to break our devices. We assume:

- that an attacker can compromise the OS and gain privileges. OSes are considered untrusted and may have lots of software vulnerabilities.

- that external malicious peripherals can connect to the device physically or remotely and can compromise the device.

- that an attacker can take control of any existing peripheral that can read/write the system memory, directly or indirectly, using an interconnect bus like TileLink [87,89]. The attacker can have access to debug ports, and they can mount passive attacks like DMA probing. We also assume that the attacker does not perform hardware attacks or software attacks that exploit hardware bugs like fault-injection attacks.

- that all communications are untrustworthy and that attackers can eavesdrop traffic and inject malicious code or data. Therefore, all incoming data is considered untrusted.

2.3 Goals to secure a lightweight device

The main aim of this work is to develop a new co-design to provide an ideal isolation and attestation architecture for lightweight devices. In this respect, our work belongs to the same family as Intel SGX [26], TrustZone [100], TrustLite [48] and the likes [18,27,30,32,36,53,57,59,60]. These solutions differ by:

Architecture type: in literature, there are three types of architectures, hardware-only architectures, software-only architectures, and hardware/software architectures that are called hybrid.

Components number and size: each architecture can be composed of one or multiple components, the number and the size of components matter because they impact the attack surface, the cost, and the performance.

Separation technique: separation techniques have an impact on the protection level, the cost and performance. Each architecture offers a separation model depending on their objectives and uses.

Attestation process: not only there are multiple techniques to achieve attestation, but also different objectives. Some architectures target static attestation, where applications are attested at system start. Others target dynamic attestation to be able to verify data and code integrity at runtime and detect injected malwares.

2.3.1 Properties for an effective isolation and attestation architecture

To develop an effective isolation and attestation architecture for lightweight devices, we focused the first part of this work in exploring architectures in literature and studying their strengths and weaknesses to define objectives required to provide a highly secure architecture that can be adapted to different targets while guaranteeing good performance and small cost. Our objectives can fall into seven points:

- **Multi-layer Isolation:** As the old saying goes, *divide to conquer*, separation has to be on multiple levels. After analysing security weaknesses, three layers of separation were identified. The first layer is the Masters' separation. Figure 2-4 illustrates a visual of memory protection. Each column represents a protection domain. Here the protection domain is a given Master. Each Master can be prohibited or not from accessing memory regions. In this thesis, we call Master a peripheral with direct access to memory, i.e. an IO Peripheral. There can be different policies to manage protection domains. The second layer is the strong isolation of the security monitor from the rest. The security monitor is the software responsible for configuring the protection hardware components, and thus, is critical for the system security. Therefore, we need to make sure it is well protected from the rest of the software. Finally, the third layer of separation is the application separation. At this layer, we isolate applications and libraries so flaws cannot propagate easily from one application or library to another.

- **Local and Remote Attestation:** Isolation alone is not sufficient. Our architecture needs to be able tu run authentic software and process trusted data. Attestation offers a way to guarantee the integrity and authenticity of code and data. The security monitor can be used to attest a code or some data but also, before communicating with other devices, they can prove their integrity and verify their software states.

- **Security Monitor Protection and Upgradability:** The security monitor has access to all memories and configures protection domains. Breaking into the security monitor is fatal to the micro-controller security. We must guarantee good protection for the security monitor. The other important point is the ability to upgrade the monitor securely to renew its security.

- **Confidentiality:** The architecture has to ensure the confidentiality of some sensitive code and data.

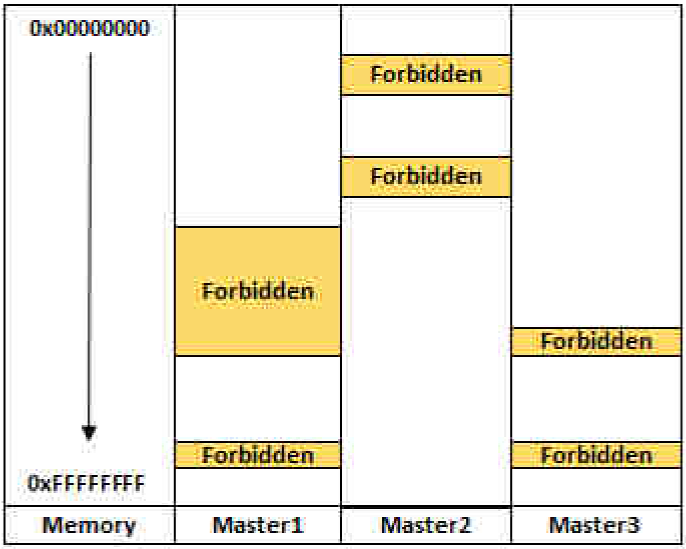


Figure 2-4: A visual of how a single contiguous memory space is segmented between different Masters.

- **Cost and Performances:** The footprint of the added hardware and software, and the runtime overhead have to be small and negligible while guaranteeing a good level of protection.

- **Flexibility:** The architecture has to be flexible. Flexibility in this thesis means that the architecture has to be fully configurable and modular. So it can be tailored to the device needs.

- **Ease-of-use:** Last but not least, the architecture integration and usage have to be easy for both hardware designers and software developers.

2.3.2 Use Cases

This part presents some use cases and techniques that can be achieved using Toubkal.

Hardware and Software Isolation: This is the main objective of Toubkal. Toubkal aims to create a real separation between different components: hardware and software.

Local and Remote Attestation: With the security monitor services, applications (and devices for remote) can verify the authenticity of other applications before starting exchanging information.

Measurements Attestation: Other than code attestation, some applications need to verify the authenticity of the data read from external peripherals. For example, imagine an insulin device that returns compromised data to a doctor about a patient. This might result in the wrong prescription and can lead to catastrophic issues. The security monitor can be used to attest data before the doctor reads it.

Algorithm Protection: Industrials have sometimes algorithms they want to keep secret from the client. A cheap way to prevent other parties from analysing the secret algorithms is to use Toubkal and prevent all non-allowed accesses. The algorithm will be only executable, and the debugger is deactivated during its execution. Nevertheless, some side-channel attacks, which are out-of-the scope of this thesis, can extract code.

Secure Firmware: The ROM is cheaper than the FLASH. However, the ROM has a disadvantage of being unable to change the code once burned. The ROM code can be large, and testing does not guarantee a code that is free from bugs and security holes, sometimes even neglected. So, most issues are discovered after fabrication which is too late to fix for running devices. Loading the Firmware into a Flash-like memory is not secure. To keep things secure, we rely on the hardware and the ROM to store a simple and small code to verify the authenticity of the Firmware, which can be stored in any other memory, at boot time or every time it is called. We call it the ***root***, and it is the immutable part of the security monitor.

Efficient Core-Crypto Engine data communication: Consider a device with three Masters: a CPU, a DMA, and a Cryptography Engine (CE). The CE can access the SRAM to read/write values for intermediate cryptographic operations. While usually the Keys are protected, an attacker can take advantage of a vulnerability like a buffer overflow or uses the DMA [75, 95] to access the intermediate cryptographic results. This way, the attacker can find the values of the keys from intermediate operations. Toubkal prevents the CPU and the DMA from reading the concerned memory region while the CE is processing its cryptographic operations. Finally, the same memory can be used by the concerned application to retrieve the result, which can be the encrypted text or the plain one.

2.4 Summary

In this chapter, we defined the targeted devices and the challenges they face. The so-called lightweight devices are systems with simple memory management. Therefore, they do not need an MMU. Lightweight devices face multiple challenges to offer an acceptable level of security:

challenges like the cost of protection mechanisms, the lack of separation, hard verification, and establishing trust between applications and devices. Then, we studied software and IO-based attacks to understand the threat we need to tackle.

Then, we presented goals and properties we aim to incorporate in our architecture. The major goals of the architecture are :*multi-layer isolation*, *remote and local attestation*, *upgradability*, *confidentiality*, *small size with a negligible run-time overhead* and *ease-of-use*. Then, we defined the boundaries of this context, for example, hardware-based attacks like side-channel and Fault attacks that are out of the scope of this thesis.

In the next chapter, we present multiple isolation and attestation architectures. Then, we discuss their security features and show the variety of techniques that were used in the literature.

Chapter 3

Isolation and Attestation Architectures

The main goal of this part is to study the existing solutions in both academia and industry and to discuss their security features. Thus, we present a detailed description of fifteen isolation and attestation architectures, and we compare the different services they provide to enforce protection and establish trust in devices. These architectures target multiple types of devices: lightweight and high-computing devices.

3.1 Background

Isolation: By isolation, we mean mechanisms that provide compartmentalization of software components. Compartments are separated and protected by a hardware component to prevent the propagation of flaws from one compartment to the others. In our case, isolation is achieved with the MPU. This includes another layer of memory security by limiting compartments from accessing any memory address.

Trusted Computing Base (TCB): The TCB consists of a set of software and hardware components that are critical for device security, like for example, MMUs, MPUs, and hypervisors. The TCB design must be thought carefully to guarantee a good security level. The TCB should be kept as small as possible to reduce the surface that is exposed to attacks and to minimize bugs and weaknesses that could be used to break the system security. Architectures can offer hardware-only TCB, software-only TCB, and hybrid TCB. Each type has its benefits and drawbacks, like for example, protection level, hardware footprint, and performance.

Root-of-Trust (RoT): The RoT is a set of gadgets that are always trusted by the running software. They are, most of the time, used to establish trust in other software components. A Dynamic RoT (DRoT) is responsible for attesting and verifying the state at runtime of any software component before the latter is executed. Admittedly DRoT offers flexibility and can be used to detect malware at runtime, but it requires more protection to avoid attacks like Time-of-Check-Time-of-Use (TOCTOU).

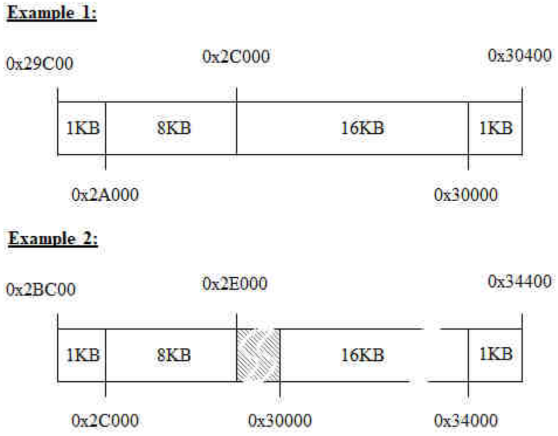


Figure 3-1: MPU configuration: The MPU in the ARMv7-M architectures requires that the memory region size must be aligned to a start address. This address is a multiple of the region size, and the region size must be a power of two. In Example 2, we can see the hole between the second and the third region.

Memory Protection Modules: Currently, most devices have a memory protection module. It is a hardware bloc interfaced with the CPU and responsible for controlling the CPU memory accesses. It allows only software components with high privileges, like a kernel, to define memory regions and attribute memory access permissions to each region. The memory protection module is used to restrict some memory regions to the software running under user mode. In industry and literature, there are different versions. For example, we have ARMv7 Memory Protection Unit (MPU) [12], ARMv8 MPU [101], the RISC-V Physical Memory Protection (PMP) [88], the Intel Execution Aware-MPU (EA-MPU) [71], the Mondriaan Memory Protection (MoMP) [103] and the tailored-MPU [94]. However, depending on the MPU version, there are some limitations. For example, for an *ARMv7-MPU*, which is present on some ARM Cortex-M devices, as shown in figure 3-1, the base address of a region must be aligned to its size, and the size must be a power of two. These limitations were overcome in the ARMv8 MPU, thus, offering developers a finer granularity which makes defining memory regions easier.

The *RISCV PMP [88]* is very similar to the ARMv8 MPU. There are few differences, such as the lock bit. The lock bit is used when we want to lock the configuration of the memory region and also to apply that restriction for the privileged mode and not only the user mode. Once a memory region is locked, it cannot be changed until the system reset.

The *Intel EA-MPU [71]* is very different from the previous modules. It is a Program Counter (PC) based memory protection. Developers have to define for each instruction range the memory regions they have access to.

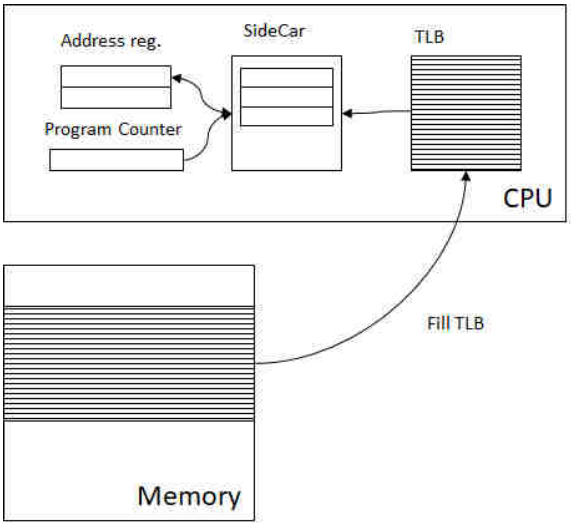


Figure 3-2: MoMP architecture: It is mainly composed of a Translation Look-aside Buffer (TLB), sidecar registers, and a reserved memory area, called Permission Table, to store all memory regions configuration. When the processor tries to access a memory region, it checks within the sidecar register if there is a match or not. If there is no match or the permissions are not valid, MoMP tries to reload the sidecar from the TLB. If the TLB does not have the permissions configuration, the MoMP tries to look them up from the reserved memory area storing all configurations.

The *Mondriaan Memory Protection (MoMP) [103]* (see Figure 3-2) looks more like a simplified Memory Management Unit (MMU). It incorporates a Translation Look-aside Buffer (TLB) like mechanism. MoMP is a very fine-grained protection module, and contrary to the previous modules, it offers multi-protection domains with shared memory. MoMP can define for each word access permissions, and to reduce memory and run-time overhead, MoMP proposes a Run-Length

Encoding (RLE) algorithm to compress permissions for a group of words. MoMP also offers virtual addressing if needed. To check memory accesses permissions, the MoMP uses sidecar register first for a fast access. If the permissions are not found or not valid, the MoMP attempts to reload the sidecar registers from the TLB. If, again, the permissions are not found/valid within the TLB, the MoMP tries to reload the TLB from the permission table stored the memory. Finally, if the permissions are not found/valid, then it is an unauthorized access.

The *tailored-MPU [94]* for low-power micro-controllers is very similar to the MoMP. However, tailored-MPU is more straightforward and offers fewer features. Because of the minimal cells of low-power micro-controllers, tailored-MPU has to reduce its footprint and implement simple mechanisms which, unfortunately, add some run-time overhead.

There is a limitation that all the previous memory protection modules suffer from. They only monitor memory accesses between the core and peripherals. IO peripherals (e.g., DMA) are out of reach to the MPU. Therefore, a compartment having access to the DMA can potentially read/write anywhere in memory.

3.2 Security Properties

The studied architectures propose different security features. To build a comparison between these architectures, we will preview all the mechanisms found in at least one architecture. These mechanisms are supposed to guarantee strong software isolation.

Inter-Process Communication (IPC): IPC refers to mechanisms provided by an OS to processes so they can communicate with each other and share data. In Trusted Computing, IPC mechanisms must make communication between two processes secure and non-transparent.

Attestation: In order to guarantee strong security, the device should support attestation to verify the authenticity and integrity of code or data state. Trusted computing architectures may offer attestation to establish trust in a specific code or data. There are two types of attestation; local and remote attestation. Local attestation is when a piece of code attests another one embedded on the same device. Remote attestation is when a piece of code attests another one embedded on a different device.

Roots of Trust (RoT): RoTs are a set of hardware and software components that are inherently trusted. They perform critical operations like measuring and verifying the software, protecting cryptographic keys, and performing device authentication.

Dynamic application loading: Dynamic application loading is the ability to load, update, and delete applications at runtime.

Application reboot: Application reboot mechanism allows the device to reboot a specific application after being compromised, for example, without requiring rebooting the whole system.

Exception handling: Exception handling mechanism makes sure that when an exception rises, it does not lead to any leakage of information.

Code confidentiality: Code confidentiality is a technique used to ensure sensitive static data or software Intellectual Property (IP) cannot be leaked with untrusted parties. Encryption is usually used to hide sensitive information and protect it from attacks such as Side Channel attacks.

3.3 Architectures

During this thesis, we studied multiple isolation and attestation architectures. Architectures that helped us draw weaknesses and strengths to tackle and to improve. Even if this thesis targets lightweight architectures, we also studied non-lightweight ones because they may offer interesting isolation and attestation techniques that can be adapted for lightweight devices with a small cost and good performance. This section presents the most popular ones. We present detailed descriptions of fifteen architectures. We varied the selection of architectures, and we did not limit our study to academic architectures only, we looked into industrial solutions too. The architectures were ordered chronologically by date.

3.3.1 Mondrix

Mondrix [104] is an extension of Linux OS using the MoMP memory protection module 3-2 previously presented. The main goal of Mondrix is to enforce the isolation between software activities. Mondrix provides cross-domain calling guaranteeing that a thread (caller) can access another thread (callee) domain only from pre-defined entry points the callee has white-listed the caller. Cross-domain calling also guarantees that a caller returning from a cross-domain calling returns only to the instruction just after the cross-domain call.

Mondrix provides a *memory supervisor* to facilitate integration. It can be added easily under an existing kernel. The memory supervisor is composed of two layers, a top layer and a bottom one.

The bottom layer is responsible for writing permissions tables in the memory. While the top layer is responsible for providing specific APIs for the kernel to allocate memory, it is responsible for revoking permissions when, for example, a memory region is freed. It also keeps track of domains' permissions to access memory.

3.3.2 SMART

SMART [30], see Figure 3-3, is a lightweight solution offering dynamic RoT for low-end devices. SMART is a hybrid approach based on minor hardware changes and a small immutable software to implement a minimalist way to remote attestation [37]. SMART prototypes were developed for the ATmega103 and the openMSP430.

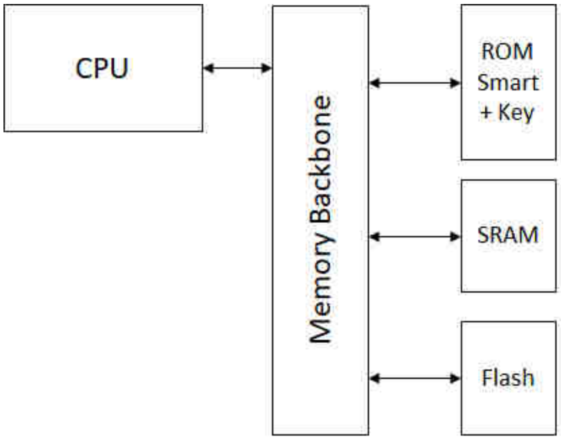


Figure 3-3: SMART architecture: SMART is composed of the memory backbone, and the ROM to store immutable code and a cryptographic key. The memory back bone controls all memory accesses, its primary goal is to block any attempt to retrieve the cryptographic key or run the ROM code in an unexpected way.

SMART requires some features to work correctly. First, we need a ROM, secure storage for the cryptographic key, a control of micro-controller accesses to the ROM and the secure storage, and the ability to reset the device and erase memory. Some assumptions, such as the adversary cannot tamper with the ROM. All other IO peripherals are disabled while SMART is processing. They have neither access to the ROM.

SMART provides remote attestation of a defined memory region by the verifier. SMART is executed by the prover and attests the memory region then jumps to it. SMART sends a proof of execution to the verifier. To attest a memory region, SMART computes the corresponding HMAC and sends the result to the verifier. The verifier calculates the HMAC for the same region then compares both HMACs.

The cryptographic key is stored within the ROM. The only code, having access to the key, is SMART code stored within the same memory. Key protection is PC-based. To avoid ROP attacks, SMART enforces ROM code protection with a single point entry and a single last point. This ensures that the ROM code will be called from one entry, and run until the end of the code. The key is used to calculate the HMAC of a given memory region. SMART disables interrupts during HMAC calculation.

The ROM code is tiny and was safely written. When SMART finishes its operations, it cleans all sensitive related data. If SMART got interrupted somehow, the clean-up might be skipped. Therefore, the hardware, after reset, erases the whole memory to ensure there is nothing sensitive left.

3.3.3 Sancus

Sancus [67] main goal is to design a hardware-only low-cost security architecture. Sancus features three main points: isolation of software components, secure communication and attestation, and a zero software TCB.

Sancus targets lightweight devices in a large network. Theses devices are called *nodes*. The main challenge of Sancus is to be able to run a secure multi-application *node* without trusting a single piece of code. A group of *nodes* is managed by an infrastructure provider (IP), and they share the same symmetric Key K_N. This key is hard-coded so it cannot be leaked easily from the software.

To load an application within a *node*, Sancus uses software providers (SP). Each SP has a unique ID. To load securely an application, SP uses a derived key $K_{N,SP}$ from K_N and SP. The same key can be generated in the *node* as it knows the ID of the SP and the K_N.

Sancus can have unprotected applications, and protected applications called software modules (SM). As figure 3-4 shows, each SM has an entry point, a text section, and a protected data section. An SM can also use an unprotected area to minimize the size of protected memory areas. Each SM derives a unique key $K_{N,\ SP,\ SM}$ from K_N and the SM ID. The number and the size of SMs are static and can be chosen during Sancus synthesis.

Sancus opts for a program counter (PC) based protection. The design was mainly based on

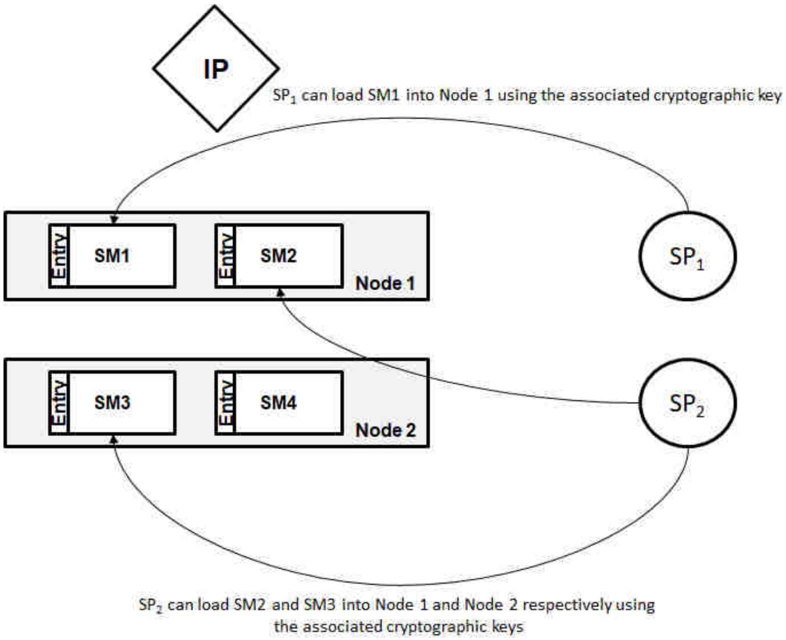

IP
SP$_1$ can load SM1 into Node 1 using the associated cryptographic key
Entry
SM1
Entry
SM2
Node 1
SP$_1$
Entry
SM3
Entry
SM4
Node 2
SP$_2$
SP$_2$ can load SM2 and SM3 into Node 1 and Node 2 respectively using the associated cryptographic keys

Figure 3-4: Sancus model: The infrastructure provider (IP) provides multiple nodes (in grey). Each node can contain multiple protected software modules (SM). Sancus uses software providers (SP) to load SMs within the nodes. Each SP and node can derive a cryptographic key to load protected SMs. An SM is mainly composed of an entry point and code and data memory regions. The SMs have to be called from their entry point.

the work in [97] because it was proven as adequate protection against kernel-level or process-level malware [96]. Furthermore, this mechanism is also used to enforce attestation process protection. To compute a Message Authetication Code (MAC), Sancus offers an instruction. When the instruction is called from within the SM, Sancus automatically uses the right key based on the address of the PC.

Sancus introduced five new instructions. The `MAC_seal` and `MAC_verify` are used respectively to calculate a MAC with the key $K_{N,\ SP,\ SM}$ over a given memory range, and to calculate the MAC of a specified module with the current module's key, then it verifies if it matches with the pre-calculated MAC stored in memory. The `get_id` is used for secure linking. To avoid the overhead of MAC processing, each module is assigned a unique key during load time. The instruction can be used then to retrieve the module key and will be used for later loading.

The other two instructions are `protect` and `unprotect`. They can only be called from within

SMs, and are used to enable or disable all the protection mechanisms respectively.

3.3.4 Software Guard eXtention

The Intel Software Guard eXtention (SGX) [5,42,61] offers Trusted Computing to a high computing device. SGX implements protection for both applications' code and data. SGX offers dynamic RoT and can guarantee authenticity and integrity of memory chunks, remotely and locally.

SGX offers secure containers which contain protected application. These containers are called enclaves. SGX trusts only a microcode and a privileged enclave. All the rest of the software is untrusted (e.g., OS, hypervisor, firmware). Whether it is an enclave or a normal process, they share the same address translation rules. It means that nor the TLBs nor caches were extended, and secure enclaves can be managed by the OS and the hypervisor. This makes SGX compatible with existing systems with an MMU. Nonetheless, SGX always ensures that every core's TLB contains data of the running container. SGX ensures that the core TLB is flushed for every switch context between enclaves and normal processes.

SGX microcode is the one responsible for ensuring the authenticity and confidentiality of containers pages. SGX uses an extended version of Merkel tree [15,63] where the OS can dynamically shape the tree.

Enclaves are created and initialized by untrusted software. During initialization, all the enclave code and static data are attested, and they cannot be changed. SGX software's attestation uses the Intel's Enhanced Privacy ID (EPID) groups signature [19]. To avoid colossal cell area overhead, SGX uses private enclaves with direct access to SGX keys. These private enclaves are signed with an Intel private key whose public key is hard-coded within SGX and accessible by the microcode. This approach is not very convenient for many vendors. Intel and only Intel can sign the private enclave. Intel also decides which enclaves can run.

According to [26], Intel presumes that SGX guarantees DRAM authenticity and confidentiality using the Memory Encryption Engine (MEE) [90].

3.3.5 TyTAN

TyTAN [18] is a security architecture for small devices. It is based on an Intel's Siskiyou Peak architecture, and a modified FreeRTOS. It provides dynamic loading and configuration of secure tasks, secure IPC, with real-time guarantees.

Figure 3-5 presents the architecture of TyTAN and its different components. There are two

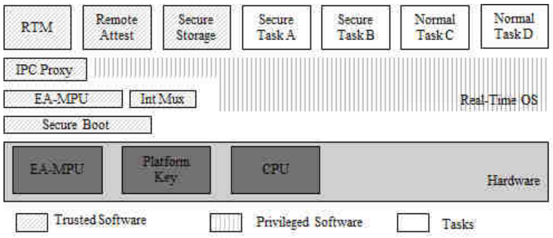


Figure 3-5: TyTAN architecture

different types of tasks: Secure tasks and Normal tasks. Normal tasks are isolated from secure tasks. Secure tasks are isolated from each other and from other software components. Their memory regions are protected using the Execution-Aware MPU (EA-MPU). Tasks can be loaded dynamically using an ELF loader.

The TCB is composed of many parts, such as a secure boot, a Root of Trust Measurement (RTM) task, an EA-MPU driver... The secure boot is responsible for loading all other parts of the TCB to ensure their integrity. The RTM task is used for the tasks' integrity: it computes a cryptographic hash of the binary task code that represents the identity of the task. This calculus is the basis for local and remote attestation. For local attestation, task identity can be used to attest the task. For remote attestation, the Remote Attest task uses a MAC with an attestation key to prove the authenticity of a task identity to a remote verifier.

The EA-MPU driver is responsible for configuring the EA-MPU dynamically at each load or unload of application. The isolation is based on the Program Counter (PC). The EA-MPU driver configures the eighteen slots with the memory regions and their access control rules.

Tasks communicate with each other using an IPC proxy task. This one allows transmitting a message m from a sender task S to a receiver task R. For small messages, S copies m and idR (identity of task R) into the CPU registers and calls the IPC task via a software interrupt (SWI). The IPC task using idR determines R memory region and writes m and idS. For large messages the IPC task creates a shared memory region between the involved parts.

TyTAN offers secure storage where tasks can store their sensitive data. It is based on data encryption. For each task, a key is derived from its identity. The key is used to encrypt data before storing it into the secure memory. The uniqueness of the task identity guarantees the uniqueness of the encryption/decryption key.

TyTAN also offers an interrupt multiplexer task used to save task context securely, and clear

Central Processing Unit (CPU) registers before the interrupt handler takes control. There is an interrupt descriptor table (IDT) protected by the EA-MPU where handlers for interrupts are determined.

The main objective of TyTAN is to ensure the integrity of critical tasks while maintaining the real-time criteria in low-power micro-controllers. This is done through the secure boot and the EA-MPU.

3.3.6 TrustLite

Just like TyTAN [48], TrustLite is based on Intel's Siskiyou Peak architecture and uses a very modified EA-MPU. Unlike TyTAN, it is a generic Protected Module Architecture (PMA), OS-independent. TrustLite provides compartmentalization and guarantees code and data integrity and confidentiality of the compartments.

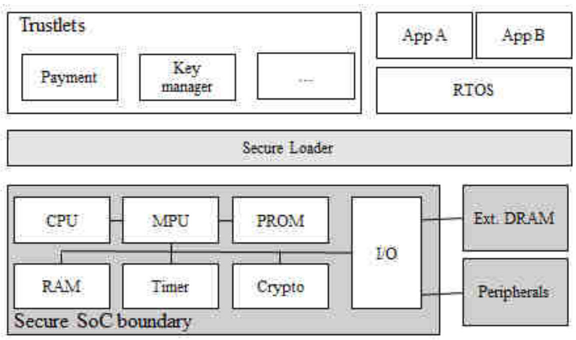


Figure 3-6: TrustLite architecture

TrustLite compartments are called *trustlets* (See figure 3-6). A trustlet can contain many software components. It is defined by its code, data, and other memory regions and an entry vector. This entry vector contains one or more entry points that allow other tasks and *trustlets* to execute authorized functions of the corresponding trustlet.

The EA-MPU has 32 protection regions and for each trustlet, we declare multiple different regions. These memory regions are only accessible by the corresponding trustlet. The EA-MPU gives a trustlet access or not to a memory region depending on the PC, just like TyTAN. If the PC is inside the trustlet code, it has access to the authorized memory regions; if not, an access violation exception is raised.

TrustLite also handles interrupts. Its interrupt manager is called the Exception engine. To ensure that no information is leaked while interrupting a trustlet at any time, the Exception engine stores the CPU state to the current trustlet stack, it stores the stack pointer in the trustlet table, and it clears all general-purpose registers. After executing the interrupt handler, the interrupted trustlet resumes running by jumping to its entry point and restoring its stack as quickly as possible.

TrustLite provides a protected IPC and an unprotected IPC. In TrustLite, signaling and short messages are done in the form of an RPC. Concerning large buffers, *trustlets* use a shared memory defined in advance in an EA-MPU region. The size of the page and the participants are defined. Protected IPC uses a simple handshake protocol to attest the identity of the receiver and sender, and to create a cryptographic session key used to authenticate messages in both directions.

3.3.7 uVisor

uVisor [9] is an architecture for lightweight devices, an open-source software hypervisor which creates secure, separate domains called boxes, on Arm Cortex-M3/M4/M23/M33 micro-controllers, compatible only with Arm Mbed^{TM} OS. The main objective of uVisor is to provide strong isolation between boxes and protect sensitive data from being leaked. Its TCB is composed of both software and hardware components.

uVisor is based on an Arm MPU, a trusted hardware component in the TCB. It provides compartmentalization by protecting each box memory area and restricting access to peripherals. The MPU guarantees that only authorized boxes can access a specific memory region. In the case of non-authorized access, a memory fault is generated.

A box represents a process, and for each one, uVisor allocates a Heap and a Stack and a static Stack (See figure 3-7). It stores within the static Stack the box context (sensitive global data only accessible from within the box). For each box, there is a constant Access Control List (ACL) where authorized hardware peripherals and memories are defined for the box.

The TCB is composed of several secure software components. The virtual MPU (vMPU) configures the MPU upon the box switch. An Arm MPU has only eight configurable regions. Therefore, thanks to the vMPU, we can declare more than eight regions for each box. When the OS switches to a box, uVisor reconfigures the MPU with the first eight regions from the ACL. Then, whenever there is an attempt to access a region that was not configured, the MPU will generate an error. The vMPU will retrieve the fault address and check within the box ACL if there is any region that contains that address. If it is the case, uVisor reconfigures the MPU with the right region and

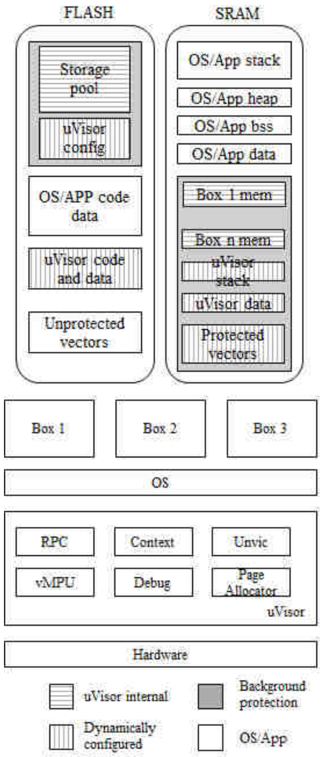

FLASH
SRAM
Storage pool
uVisor config
OS/APP code data
uVisor code and data
Unprotected vectors
OS/App stack
OS/App heap
OS/App bss
OS/App data
Box 1 mem
Box n mem
uVisor stack
uVisor data
Protected vectors
Box 1
Box 2
Box 3
OS
RPC
Context
Unvic
vMPU
Debug
Page Allocator
uVisor
Hardware
uVisor internal
Background protection
Dynamically configured
OS/App

Figure 3-7: uVisor architecture

recovers from the fault. If it is not, it is, clearly, non-authorized access.

uVisor provides a debug component to help users to identify problems and failures encountered during application execution. The user can create a debug box and customize system reaction to faults.

The Remote Procedure Call (RPC) component allows a box to call functions that need to be executed in the context of another box. By default, boxes are unauthorized to call other boxes' functions that compute sensitive data. However, uVisor can declare RPC gateways to permit some functions to be called by other boxes. There are two types of gateways: synchronous and asynchronous.

The Page Allocator distributes pages and protects access to them. During uVisor initialization, it allocates heap memory into memory pages with the same size. A box can request pages from the Page Allocator, and if they are available, the Page Allocator secures the access for the box. Then the box can securely allocate memory inside the pages.

The Unvic component manages interrupts. Interrupts registration is exclusive to only one box. Registration is based on a first-come, first-served basis. Other boxes can use an interrupt when it is freed by its owner. When interrupts occur, uVisor protects from data leakage between two domains by saving and clearing all critical registers. Then it forwards them to their unprivileged handlers. These handlers are executed in the context of the owner box.

The context component is responsible for the context switch. This component is called by the OS, or by the Unvic component, or by the RPC one. This module calls the vMPU to reconfigure the MPU with the right configuration. In case it is called by the Unvic or RPC components, it also stores the stack state of the source box and sets the state of the destination box.

3.3.8 TockOS

TockOS [53] in an open-source OS dedicated to Arm Cortex-M3/M4 micro-controllers. It achieves isolation and memory protection and is compatible with different languages. Tock is written in Rust [54], a programming language like C++ that provides better memory safety (no buffer overflow, no double frees). Its memory efficiency and performance are close to those of C++. Tock is also based on the Arm MPU to provide compartmentalization and memory protection.

TockOS's kernel (Figure 3-8, bottom side) is isolated from applications. The kernel is composed of two parts: a trusted one called the core that is responsible for critical tasks of the OS like scheduling, and a non-trusted one called capsules, that contains peripheral drivers and non-system-critical tasks. Although capsules are non-trusted, they run in the privileged mode and can communicate with the core and with each other to achieve their task. Rust's type system guarantees isolation between capsules.

Applications are processes (Figure 3-8, upper side), they are untrusted, and they run unprivileged. They are isolated from each other, and each one has its memory region protected by the MPU from non-authorized accesses like, for example, from other processes. If a process crashes, it does not make the system crash; it can restart without interfering with the kernel. It is also possible to load processes at runtime without restarting the kernel. Processes can communicate with each other via Inter-Process Communication (IPC), and with the kernel through system calls.

Although processes can be written in any programming language that supports Position Independent Code (PIC) and the Cortex-M architecture, C and C++ are the only languages that are widespread in writing such applications.

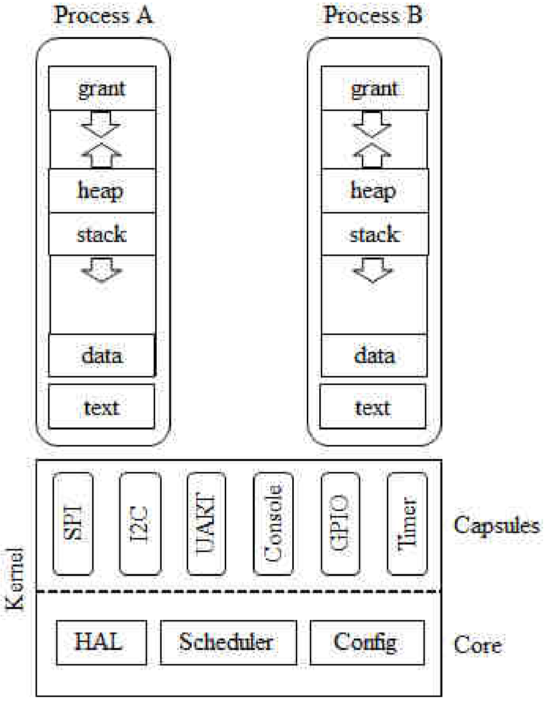

Process A
grant
heap
stack
data
text
Process B
grant
heap
stack
data
text
SPI
I2C
UART
Console
GPIO
Timer
Capsules
Kernel
HAL
Scheduler
Config
Core

Figure 3-8: TockOS architecture

System Calls allow unprivileged applications to communicate with the kernel. Tock provides 5 system calls: *command*, *allow*, *subscribe*, *memop*, and *yield*.

The *command* system call tells capsules to execute a specific action synchronously. It takes four word-sized entries. The first entry tells the kernel which capsule should run the task. The second one specifies the requested command. The third one provides more fine-grained actions, and the last one contains the caller identifier. Command should not be long to execute; however, commands can start an asynchronous task via a subscribe system call.

The *allow* system call is like the command system call, but, it is used when there is a large buffer to transfer. It defines a memory region as shared between a capsule and an application. It takes four arguments: The first one specifies which capsule will be granted access. The second one determines the purpose of the call. The third and the fourth ones take a pointer to the start address and its size, respectively.

The *subscribe* system call sets up callback functions for capsules to be run in response to an event. It takes as arguments the capsule number and a callback function pointer. Once the event is triggered, the callback is fired.

The *memop* system call invokes the core to expand the memory available to the process.

The *yield* system call pauses process execution until the callback completes.

TockOS also provides a kernel abstraction called *grants*. They allow capsules to have access to an allocated memory region within each process. This memory region is not accessible when a process dies. Capsules will need to allocate memory to execute processes demands, as such demands that cannot be anticipated in advance. So, rather than allocating resources statically and wasting lots of memory, grants solve the problem as capsules can dynamically allocate memory from the grant area within each process, and the grant area cohabits with the process heap area.

3.3.9 Sanctum

Sanctum [27] is a RISC-V processor prototype that combines minimal hardware modification with a secure software component to provide Isolation. It is inspired by Intel SGX (Software Guard Execution). Unlike SGX, Sanctum protects against some side-channel attacks, like page fault address and cache timing.

Sanctum offers software isolation at an enclave granularity level. Each enclave controls and manages its page table and page faults and have a separate DRAM region corresponding to distinct sets in the shared Last-Level Cache (LLC). This ensures the protection against Software side-channel attacks and cache timing attacks.

Sanctum uses a security monitor(SM) to manage enclaves (creation, destruction, and access) and interruptions. It is a part of the TCB. The security monitor integrity is checked during the system's boot; the ROM code is executed and calculates the hash of the security monitor. This hash is used to attest SM's identity. Then the security monitor takes control, and provides an API to create and destroy enclaves, and manage switches between enclaves.

When Sanctum's security monitor catches an interrupt, it saves core registers of the running enclave, then cleans them before exiting the enclave. The interrupt handler is then invoked and executed. After handling the interrupt, the enclave resumes execution and the security monitor restores the enclave's state.

Sanctum uses a slightly modified MMU; the modification consists of adding another page table base register so that we can have one for untrusted code and the other one is for the running enclave, and only the SM has access to those registers.

3.3.10 TrustZone-M

TrustZone [100] is a hardware-only security architecture proposed to Systems-on-Chips to enforce device security. It was first designed for the Cortex-A processors, especially for mobile phones. They wanted to secure phones to the carriers so we could not purchase a phone from a carrier and take it across another one. TrustZone provides a two worlds security model, dividing the system into two separate environments, a secure one and a non-secure one. TrustZone was extended, a few years ago, to the Cortex-M processors family [72, 101]. Both TrustZone versions, for Cortex-A and Cortex-M, share the same global objective. However, they differ in some important technical details. This is because the Cortex-M family targets real-time devices with minimal resources, and they have to be deterministic. Therefore, Arm engineers made some changes to make the TrustZone technology in line with lightweight device constraints. We call the TrustZone for the Cortex-M family TrustZone-M. Here, we focus more on the TrustZone-M.

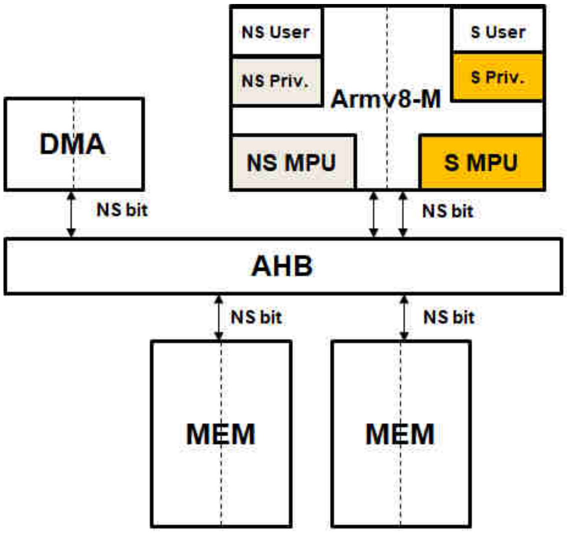


Figure 3-9: TrustZone-M architecture

TrustZone-M (see Figure 3-9) was designed for devices with low-power consumption, limited resources, and real-time processing with deterministic behaviour. Therefore, TrustZone-M limited the interaction with the software. In other words, most of its mechanisms are done at the hardware level instead, unlike TrustZone-A, where many mechanisms were hybrid.

First of all, the separation between the secure world and the non-secure world is memory-mapped. An application running from the secure memory region means the processor is in the secure state, and vice-versa. In TrustZone-M, there are multiple entry points, comparing to TrustZone-A where there is only one entry point, and it is the secure monitor handler.

TrustZone-M introduces three new instructions to support multiple entry points. The secure gateway `SG` instruction is used to switch from Non-secure to secure world. `SG` must be called at the first instruction of the secure entry point. This way, it prevents non-secure code from calling invalid secure entry points. The branch with exchange to non-secure state `BXNS` branches or returns to the non-secure world. The branch with link and exchange to non-secure world `BLXNS` is used from the secure world to call a function from the non-secure world.

Transitions from a world to the other one can also happen with exceptions and interrupts. A secure or non-secure state can own each interrupt. Interrupts, whether secure or non-secure, can raise without any restriction. TrustZone-M has two stacks, a secure and non-secure one. Each stack has its physical stack pointers, main stack and process stack, to separate between both states.

Memory addresses are tagged with a bit. This bit determines if the address space is within the secure world or the non-secure world. This bit, Non-secure callable (**NSC**), propagates from the processor through the interconnection bus to all other components. When the processor is in the secure state, all components are in the secure state too, and then the whole memory is visible. However, when the processor is in the non-secure state, so the other components, and then, the only visible memory is the one tagged with the non-secure bit.

TrustZone-M can come with Trusted Firmware-M (TF-M) [11]. TF-M is a software component offering multiple services and protection mechanisms to enforce Trusted Computing. Such mechanisms are: Isolation, IPC, trusted boot, attestation, secure storage, and a cryptographic library. TF-M lives in the secure world and is responsible for device security.

3.3.11 Sopris

Sopris [43] is a new micro-controller designed with security constraints. Researchers at Microsoft designed this new micro-controller based on seven properties that they defined required for a highly secure device. The seven properties are:

Hardware-based Root of Trust: Hardware provides effective security comparing to Software, and with Physical countermeasures, the device can resist side-channel attacks to protect keys from being leaked.

Small Trusted Computing Base (TCB): The TCB should be kept small to reduce the attack surface of this critical module. The smaller is, the harder it will be for an attacker to break into the TCB.

Defense-in-Depth: This technique has been used as a military strategy that aims to apply multiple layers of protection to delay an attack and to make it expensive for an attacker. Also, it is a way to buy time and act before an attacker succeeds in compromising the system.

Compartmentalization: Or isolation of components prevents vulnerabilities in a component to propagate to other components and retrieve their secrets.

Certificate based authentication: Local or remote authentication based on certification rather than passwords to prove identities when communicating with other components/devices/servers, because they cannot be stolen or forged and prevent against man-in-the-middle attacks.

Renewable Security: No system could be qualified as invulnerable. Attacks evolve with time, and attackers find new attacks vectors, they will always find a way to break a system. A device with renewable security can be able to update and protect itself from newly discovered attacks automatically.

Failure reporting: A device should be able to report failures that occur. Reporting failures helps in improving both protecting assets and user experience.

Within Sopris, we find a whole hardware subsystem dedicated to security, and it is called Pluton. It contains a Security Processor CPU, cryptographic engines, RNG, a key store and a complex operation engine. The cryptographic engines include cryptographic algorithms like AES, SHA, RSA, and ECC. The hardware RNG is used to counter timing attacks, and it is also used for key generation and other cryptographic operations. The complex operation engine is used when we need to execute an operation that needs more than one cryptographic engine.

To create Sopris, they modified the MT7687, and they made three changes to the micro-controller; they added the Pluton subsystem module, they added an MMU to the CPU, and they increased the SRAM. An MMU instead of the MPU in a micro-controller was justified by the fact that it supports multiple levels of isolation and multiple address spaces from which an OS can create process-isolation compartments.

3.3.12 EPOXY

Recently, research has been proposing a different kind of compartmentalization solutions. Such solutions [22, 23, 47] are LLVM-based embedded compilers and offer an automated way to create two or more isolated execution environments.

[23] proposes a solution to create two separated domains for tiny embedded devices such as Amazon Dash and SD card controllers. This solution is called EPOXY and is an LLVM-based compiler that creates two separated domains based on the two privileged execution modes, user and privileged modes, and the MPU. EPOXY offers an approach to automatically identify all sensitive instructions and run them in the privileged mode. The rest of the code, non-sensitive one, is run in user mode. To limit access to non-sensitive code and separate between the two domains, EPOXY uses the MPU. Sensitive instructions are defined as instructions calling sensitive peripherals. For example, a UART, or a DMA.

EPOXY targets bare-metal applications running on an Armv7-M architecture. EPOXY's design contains four components to achieve protection: *access controls, privilege overlay, SafeStack, and diversification.*

Acess controls allow to define read, write, and execute permissions for each memory region executed or accessed from instruction run in the user mode. For this matter, EPOXY configures the MPU to limit access to all non-sensitive instructions. The developer provides EPOXY with information about the sensitive peripherals (address spaces, sizes..). Then, EPOXY uses this information alongside automatically retrieved information during the linking stage to generate the write configuration of the MPU. Finally, the compiler adds to the startup, a code to initialize the MPU.

Privilege overlay is a technique used to identify all sensitive instructions and wrap them with a mechanism that elevates the privilege, so they are run in a privileged mode. Then, the MPU is used to limit access to the user mode to create a clear separation between the two modes. To identify sensitive instructions, EPOXY uses static analysis and also source code analysis. Static analysis is used to identify restricted instructions and restricted memory accesses. Code source analysis is based on identifying annotations by the developer to find sensitive operations.

SafeStack [50] is used by EPOXY to defend against control-flow attacks. SafeStack is a protection mechanism that moves unsafe variables to a separate unsafe stack. Unsafe variables are global ones and the ones susceptible to memory out-of-bound. The researchers modified SafeStack to support bare-metal application and guarantee a low run-time overhead.

Diversification [52] is used by EPOXY for functions, data, and registers to mitigate code reuse attacks and data corruption attacks. Diversification is based on the amount of non-used memory. EPOXY uses the unused memory to spread functions and data all over the memory. Then, with SafeStack, we have multiple groups of functions or data separated with guard bands.

3.3.13 ACES

While EPOXY [23] offers a two compartments solution based on privileges, ACES (Automated Compartments for Embedded Systems) [22] offers the opportunity to have multiple compartments in bare metal applications. We start by presenting here ACES.

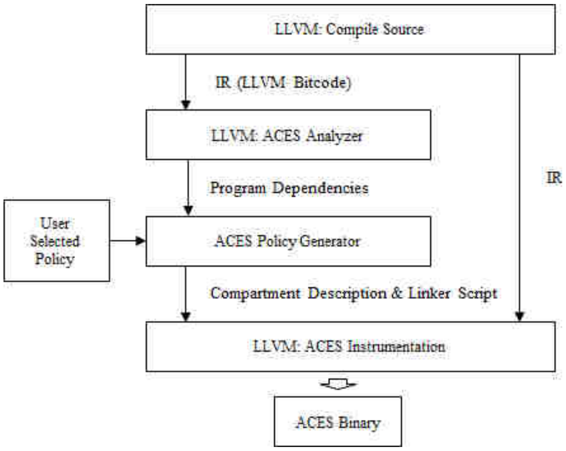

LLVM: Compile Source
IR (LLVM Bitcode)
LLVM: ACES Analyzer
Program Dependencies
IR
User Selected Policy
ACES Policy Generator
Compartment Description & Linker Script
LLVM: ACES Instrumentation
ACES Binary

Figure 3-10: Aces architecture

ACES is an LLVM-based compiler that creates automatically isolated environments in bare metal applications. ACES fulfils its objective by performing four steps (see figure 3-10): Program Analysis, Compartment Generation, Program Instrumentation, and Run-time Enforcement. The program analysis(step 1) creates the PDG (Program Dependence Graph); it captures all control-flow, variables, and peripherals dependencies. This is used to generate the region graph. The region graph is then used to generate compartments (step 2).

The user provides a compartmentalization policy. There are 3 policies: *Naïve Filename*, *Optimized Filename* and *Peripheral*. The *Naïve Filename* consists in considering each file name a compartment. Then, The *Optimized Filename* consists of merging files with the same regions' access into the same compartment. Finally, the *Peripheral* policy consists of grouping code that accesses a single peripheral into the same compartment. Then ACES applies the chosen policy.

To meet the eight regions' MPU constraint, ACES merges some regions with each other. The program instrumentation (step3) creates a compartmentalized binary. It instruments the program and identifies all compartment transitions. All cross-calls are identified, and it creates metadata. This metadata is used to validate a transition.

After instrumenting the binary, ACES lays out the program in memory to meet the MPU constraints which are the region's size are powers of two and the region's start address is aligned to its size. This routine is done in two steps: In the first round, ACES uses a linker script that ignores these constraints, the resulting binary is used to extract regions sizes. Then, in the second round, ACES uses another linker script to expand memory regions into the powers of two and lays out regions from the highest to smallest to minimize holes between regions. The resulting binary is ready for execution.

3.3.14 HEX FIVE MultiZone

MultiZone [14] is a security architecture for RISC-V based cores providing isolation for both lightweight and multi-core devices. MultiZone is composed of a software component using Rocket Chip security hardware modules to enforce device security. MultiZone does not require any change to the existing source code to adapt it to its architecture. Hence, it does not offer thread-level isolation, but, software component level. Here, a component can be an OS, a cryptographic library, or a network stack.

MultiZone uses the Physical Memory Protection (PMP) hardware module found in the Rocket Chip to create separation between these components. MultiZone software consists of the following components:

Secure Boot: A two-stage secure boot, it verifies the authenticity and integrity of the firmware image.

nanoKernel: A small software component responsible for managing separation, in other words, switching from a context to another and guaranteeing that each zone is running under the right configuration. The nanoKernel is formally verified.

Configurator: The configurator combines the linked zones and the provided memory regions access permission to generate the right PMP configurations for the nanoKernel.

Messenger: An IPC provided to exchange messages securely from a zone to another one.

HEX FIVE offers the possibility to add IO peripherals separation by adding a hardware module called IOPMP (IO for In/Out). The idea behind IOPMP is to plug it in front of each Master to control its memory accesses. The IOPMP can also be interfaced with a Slave to control memory accesses.

3.3.15 VRASED

VRASED [68] is very similar to SMART. VRASED is an extension of SMART. The main additions of VRASED, comparing to SMART, are the hybrid formal verification and controlling DMA memory access from accessing the secure storage.

VRASED claims that it is the first formally verified hardware/software solution for remote attestation. The researchers formalized remote attestation properties to define invariants that must hold during system execution. Each VRASED sub-module was verified, and if verification fails, the sub-module was redesigned. Then, once all sub-modules were verified, they verified the whole module.

VRASED also adds in consideration DMA access, where SMART fails. The limitation in SMART is that the DMA can access secret storage. VRASED takes into account this limitation and forbids DMA from accessing secret storage and the ROM code.

3.4 Comparison

This section gives a detailed comparison of the architectures presented above. Table 3.1 summarizes the features and properties of all the architectures. The comparison includes security properties, protection mechanisms, and architecture's features. We also looked for each architecture, whether it is deployed or not, open-source or not, and academic or industrial.

All the studied architectures offer memory isolation except for SMART and VRASED. The main objective of SMART and VRASED is to provide dynamic RoT, and they have established isolation of the root from the rest of the software. They are not considered isolation architecture because they do not provide applications' separation. There are two schools to provide isolation. Some use a separation based on the program counter while others use a memory protection module (MPD), such as MPU, MMU, and a privileged software to switch context from one to another. The privileged software configures MPD's registers to give access to the running application. Isolation granularity changes also. Some offer a coarse granularity, at a memory page or cache line, while others offer a fine granularity, at a thread level. However, the title for the finest granularity goes to Mondrix, where we can attribute permission at a word level. According to [85] granularity of isolation has an impact on performance; the finer the granularity, the more the run-time overhead grows. The next chapter studies in depth performances and deepens this aspect.

Concerning attestation, each architecture has its implementation. We can group them into two categories, hardware-only attestation and hybrid one. Hardware-only attestation, such as Sancus, uses a dedicated hardware module to perform attestation, during boot time or application loading. Hybrid attestation, such as SMART or SGX, uses a software component to perform attestation. They rely on a hardware module, like a memory protection module, to protect the software from being played with, and protect keys and intermediate data from being leaked. On the one hand, there are architectures offering simple symmetric protocols because they are cheaper than asymmetric protocols. For example, SMART uses a simple HMAC algorithm where it computes a hash over a chunk of data with a symmetric key. On the other hand, architectures for high computing devices offer more sophisticated protocols, like asymmetric ones with certification (e.g., SGX and Sanctum).

The TCB size also matters in this comparison. Especially their software components because they add an attack surface to the TCB. The smaller the area, the smaller the attack surface is. In this thesis, we have seen hardware-only solutions and hybrid ones. The hardware-only architectures, like Sancus, provides much stronger guarantees than hybrid ones. They are not vulnerable to software attacks. The issue with hardware-only architecture is that they require a lot of cell area. According to [68], the hybrid architecture VRASED has only 4.16% of Sancus registers and only 6.25% of Sancus Look-Up Tables (LUTs). The other downside of hardware-only TCB is that it is not upgradable. While hybrid TCBs can offer this feature. In the fifteen architectures, only three architectures have an upgradable TCB. The software component of the TCB in SMART and VRASED is immutable in the ROM, even if they are hybrid. Others, such as uVisor, TyTan, and TockOS can have their secure software tampered with. TrustZone-M TCB surface can snowball. This is because of its isolation mechanism based on two domains, and the secure world contains many pieces of code.

Only a few architectures offer code confidentiality. Such architectures are SGX and the upgraded version of Sancus called Soteria [38]. Code confidentiality ensures that sensitive code and data cannot be obtained by untrusted parties. SMART and VRASED root codes are also protected from being leaked from the outside of the root. Except for SGX, all these architectures offer partial code confidentiality in this thesis attacker model. The first thing, they do not use encryption, for example, to fully protect the code, and the second thing is that they assume that untrusted parties are only software components, although, we can have IO peripherals with full access to the memory actually.

Only EPOXY, ACES, and MultiZone do not need a software adaptation. EPOXY and ACES are automatic solutions to create isolation in bare-metal applications. They are more of compilers than architectures. They compile the code and instrument it according to a defined policy, and generate a secure application binary. MultiZone, is not a compiler like the two other ones. However, it has a tool to link the different applications and generate the right configuration of the memory protection module. The rest of the architectures requires a software adaptation. The issue with the software adaptation is that developers need to rewrite their code. So again, if they go fast to be in line with the industry pace and have no security background, they can add back-doors.

Isolation requires a separation between multiple applications embedded on the device. Nevertheless, applications may need to communicate with each other, even untrusted ones with the TCB. Architectures implement different types of IPCs. For small messages, they use the processor register for faster exchange. Other architectures implement shared memory regions where two or more applications create a page to share data.

Only a few tackles the IO peripheral issue. TrustZone provides only a binary separation for IO peripherals. When the CPU is in the secure state, IO peripherals are too in the secure state, which means they can access to everything, and when the CPU is in the non-secure state, the IO Peripheral cannot access the secure memory. HEX FIVE offers the IOPMP, which is interfaced with each Master or Slave, the issue here is in the case we have multiple Masters and we need to control their memory accesses. We will have multiple IOPMP, therefore more silicium size, more software complexity to manage all the IOPMPs. Other architectures, such as SGX, Sanctum, VRASED, and TockOS, cover the DMA issue partially. On the one hand, SGX, Sanctum, and VRASED limit DMA memory access with a memory controller to a single continuous space. On the other hand, TockOS relies on the memory safety of Rust to forbid buffer overflows and try to read other memory addresses with the DMA.

Table 3.1: Summary of the studied isolation and attestation architectures.

	Isolation	Inter-Process Communication	Root of Trust	Dynamic loading	Exception Handling	Application reboot	Attestation	Code confidentiality	IO Peripherals protection	Lightweight	Memory Protection Module	Hardware-Only TCB	Software adaptation	Open Source	Academic	Deployed	ISA
Mondrix [104]	x	x	o	o	x	o	o	o	o	o	Mondrian	o	x	o	x	o	Multiple
SMART [30]	o	o	x	o	o	o	x	o	o	x	-	o	x	o	x	o	AVR/MSP430
Sancus [67]	x	x	x	x	o	?	x	o	o	x	-	x	x	x	x	o	MSP430
SGX [61]	x	x	x	x	x	x	x	x	xo	o	MMU	o	x	o	o	x	x86_64
TyTan [18]	x	x	x	x	x	o	x	o	o	x	EA-MPU	o	x	o	x	o	Siskiyou Peak
TrustLite [48]	x	x	o	o	x	o	x	o	o	x	EA-MPU	o	x	o	x	o	Siskiyou Peak
uVisor [9]	x	x	o	o	x	o	o	o	o	x	MPU	o	x	x	o	x	Arm
TockOS [53]	x	o	o	x	x	xo	o	o	xo	x	MPU	o	x	x	x	x	Arm
Sanctum [27]	x	x	x	x	x	x	x	o	xo	o	MMU	o	x	x	x	o	RISC-V
TrustZone-M [101]	x	o	x	x	o	o	x	o	x	x	-	o	x	xo	o	x	Arm
Sopris [43]	x	o	x	o	o	?	x	o	o	x	MMU	o	x	o	o	o	Arm
EPOXY [23]	x	o	o	o	o	o	o	o	o	x	MPU	o	o	x	x	o	Arm
ACES [22]	x	o	o	o	o	o	o	o	o	x	MPU	o	o	x	x	o	Arm
MultiZone [14]	x	x	x	o	?	x	xo	o	x	x	PMP	o	o	o	o	x	RISC-V
VRASED [68]	o	o	x	o	o	o	x	o	xo	x	-	o	x	o	x	o	AVR/MSP430

x: Yes, o: No, xo: Partial, ?: NA, -: Non-relevant

3.5 Summary

The primary goal of Trusted Computing is to re-establish trust in embedded systems that users start loosing after all the attacks which hit them. Recently, Trusted Computing gained more and more interest from both academics and industrials. This resulted in an increase in new architectures.

In this chapter, we presented a detailed description of multiple architectures and gave a comparison of their features and security properties. The result of our comparison shows that we have in the literature a variety of mechanisms and options to establish Trusted Computing. All the studied architectures offer a good level of protection. However, they do not support all Trusted Computing features and properties.

They mainly differ in the general approach and the objective of their architecture. Each architecture is adequate for a type of device and a particular field. For example, if we are looking for determinism and real-time, we would avoid architectures using MMUs. If we are looking for high-performance computing and sophisticated applications, we would avoid EPOXY or ACES. Some architectures are compatible with multiple kinds of devices like TrustZone and MultiZone.

We can see that there have been lots of work in this area, but also that there is still room to improve the proposed mechanisms for Trusted Computing. In the next chapter, we will present an in-depth study of other aspects, weaknesses and strengths towards software attacks, performances, and TCB sizes, of five lightweight architectures from this list.

Chapter 4

In-depth MPU-based Architectures Comparison

In this chapter, we focus on five lightweight MPU-based architectures studied in the last chapter. We chose to study their security protection, their sizes, and their performances. The goal is to compare these architectures and identify the strengths and weaknesses of each one. The selected architectures are uVisor [9], TockOS [53], TyTAN [18], TrustLite [48] and ACES [22]. All these solutions are based on an MPU to protect memory and provide compartmentalization, and they have enough data or could be ported (open-source) to existing boards to evaluate their performance and security protection. They also provide different isolation and protection techniques. Other architectures were not chosen because they were similar to these ones, or they did not have enough data to run the study.

4.1 Comparison Criteria

In this section, we define the comparison criteria used in this study. The goal of the study is to evaluate the five MPU-based architectures on three levels. The first level is the **security protection**. Isolation and attestation architectures are not aimed at protecting against a specific attack. Instead, they limit the consequences of attacks. At this level, the objective is to study and compare the way these architectures prevent a flaw in one application to propagate to others.

The second level is the **size of the software TCB**. Here, we compare the size of the software TCB of the architectures in the number of line of codes (LOC). All the studied architectures have a TCB composed of a hardware component and a software component. We compare the size of the software part of the TCB because it is linked to the attack surface of the latter.

Finally, the third level is **Performance and Memory Footprint Evaluation**. Performance evaluation can be tricky and not very representative if we evaluate an application running on different architectures. The reason we think so is that for each architecture, developers have to think and code differently. Therefore, to be fair enough, we need to code the application differently for each architecture. So, instead of evaluating runtime of a whole application, we chose to evaluate critical part in the architectures independently from the running applications. The critical part we chose to evaluate are:

- *Process creation:* Each architecture uses different process creation schemes. This task is essential because it defines the resources of the process and the memory boundaries. It has an impact on boot time and also every time the program creates a new process.

- *MPU configuration:* MPU configuration can be dynamic (the possibility to add/modify configurations during runtime) or static (one configuration at boot time that it is not supposed to change until system reset). In the case of dynamic configuration, the architecture needs to re-configure every time the context is switched.

- *Context-Switch:* Here, we compare the full context switch. A context switch is usually done in two steps. In the first step, the scheduler saves the running context and load the next context, then, in the second step the MPU configuration of the next context is applied. This is an essential task that can have a massive impact on performance, and especially on the deterministic side because it is done every system tick.

- *Interrupts:* Interrupts handling is very critical. Each architecture manages interrupts differently. Here, we will compare the impact of each technique on the performance.

- *Writing in a peripheral register:* Embedded systems interact a lot with peripherals, and it is scarce to find an application that does not make use of the system peripherals. Because of the minimal access of applications in TockOS, we thought it would be interesting to compare between uVisor, ACES, and TockOS accessing a peripheral register.

- *MPU recovery mechanism:* This feature is only proposed by uVisor, which makes this architecture flexible comparing to the others. MPUs are hardware components, and each MPU has a limited number of slots(memory regions to configure). The number of slots is fixed in the hardware. However, uVisor offers the possibility to configure more than that number by offering a virtual MPU. We evaluate the performance of this mechanism to see the impact of exceeding the hardware number of slots.

4.2 Security Protection

The first difference to observe in the five architectures is that except for TockOS, all the others are written in C, which is an unsafe language. TockOS is written in Rust, a memory-safe language. Rust is supposed to deal with out-of-bounds pointers and with dangling pointers that are the sources of most memory attacks. So how Rust achieves memory safety? Rust relies on three features: ownership, borrowing and bounds-checking. For ownership, the compiler tracks the ownerships of each value. A value can only be used once and then, the compiler refuses to use it again. This way, Rust prevents double-free errors regularly found in C/C++ language. Concerning borrowing, since Rust has rules about having one mutable pointer to a variable at a time, it employs borrowing to offer developers the possibility to pass references. References are immutable by default. However, we can have mutable references. Mutable references are possible only if we have one mutable reference to one variable in a particular scope. This way, Rust can prevent data races at compile time. Finally, Rust offers bounds-checking, but it comes at a cost. Except for C, other languages have support of it.

However, when developers use the *unsafe* keyword, they write their code in other languages like C, for example, and those parts are prone to memory attacks. A developer can reproduce every security vulnerability found in C. And if an attacker succeeds in hijacking the *unsafe* code, the safe one can be hijacked as Song shows in his paper [91]. To date, the TockOS kernel contains *unsafe* code. Also Rust language libraries and mechanisms are not totally written in Rust. The interface is safe, but the underlying implementation is written in an *unsafe* code. It is also important to note that to date, it is impossible to write applications in Rust [56]. This means applications in TockOS are written in C/C++.

We classified vulnerabilities in section 2.2.1 into two categories: *temporal*, and *spatial* vulnerabilities. An attacker can modify code, data variable, or code pointer. This can lead to a code corruption attack, a control-flow hijack, or a data value attack [99]. For code corruption attacks, the attacker tries to modify the code into their own. But the five architectures define text sections with the ReadOnly attribute, so they prevent attackers from overwriting the code.

In the following, let's assume a system with two different compartments, A and B. The code of A is unsafe and contains many weaknesses, while B is safely written because it processes some sensitive data.

For data value attacks, the attacker tries to modify data values to change the execution path. If the attacker can exploit a buffer overflow from within compartment A and tries to reach a data value within the compartment B memory region, the MPU will raise an exception. The attacker can only read/write memory regions with the MPU R/W attributes; they can manipulate authorized memory regions. And while the MPU guarantees protection of compartments' memory regions, attackers cannot access, from a compartment, other compartments' resources.

For control-flow hijack, an attacker will try to redirect a code pointer to the address of their injected code, or to an already existing function or gadget. This is usually done by manipulating the return address after exploiting, for example, a *stack-buffer overflow* or by using indirect jumps or call instructions. None of the architectures mark their stack and heap segments as non-executable, by default, to prevent executing injected code. So, if an attacker succeeds in exploiting a buffer overflow or a dangling pointer, they can inject their code and redirect the path to their own code.

Except for ACES, developers have to change stacks and heaps attributes into *non-executable* in the other four architectures' code to prevent executing injected code. But even if they do so, attackers still can reuse existing functions and gadgets. The code can be divided into two categories: shared libraries' code and compartments' code. For the shared libraries' code, like *libc*, for example, the code is publicly accessible. Reusing their functions won't be prevented, but if an attacker tries to process data from a protected compartment, the MPU will raise an exception. The functions will be executed, but only in the context of the running compartment.

TockOS, ACES and TyTAN protect a compartment code from being executed in the context of other compartments [18, 22, 53]. In the case of TrustLite, the developers must choose if they want to authorize a compartment to execute another compartment code or not. uVisor does not provide such a protection. An attacker can call any function, but, just like shared libraries, these functions will be executed in the running context, which means that if the function processes some resources that are not authorized for the running compartment, the MPU will raise an exception.

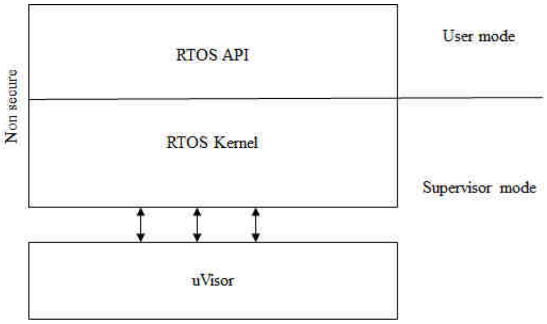


Figure 4-1: uVisor and RTOS have same privileges: RTOS is non secure, and contains many buffers, but, it has the same privileges as uVisor, which can be critical to the system protection.

Except for ACES, within all the other architectures, the OSes have the same privileges as the TCB. TockOS uses a new OS written in a safe language, while the others use traditional OSes that were developed a long time ago, did not take security measures, and use lots of buffers. This increases the chances of buffer overflows within a privileged component, and it could be critical for system security, as we will demonstrate for uVisor (see Figure 4-1).

First, note that in mbed RTOS, the `svc 0` is reserved for RTOS **SVC Handler** and it is used by the OS to switch from user mode to a privileged one. If we have a look at the **SVC Handler** function, we will find that the RTOS trusts the register `R12`, and it contains the address of the function to execute in privileged mode.

The SVC Handler can be called from any compartment by simply executing `svc 0`. Now, we take the example of the other part of the system with two compartments, A and B. In uVisor, if we have a look at how the memory is configured in the MPU, there is no protection against execution of injected code. So, from a vulnerability within A, we can inject the following simplified code:

```
LDR R12,=exploit
SVC 0

exploit:
...
```

This will load `R12` with the exploit address and then call the SVC Handler, the handler will branch to the exploit in privileged mode, and the game is over. We succeed in disabling the MPU

Table 4.1: Summary of Security protection studied in this chapter:

	Range attacks	allocation attacks	Type attacks	Code overwriting	Code protection	RAM execution
uVisor	o	o	o	x	o	o
TockOS	xo	xo	xo	x	x	o
TyTAN	o	o	o	x	x	o
TrustLite	o	o	o	x	x	o
ACES	o	o	o	x	x	x

x: Yes, o: No, xo: Partial

for example and running the system without protection.

Table 4.1 summarizes security protections studied in this paper. Except for TockOS, which is partially written in a memory-safe language and protects partially from memory corruption attacks, the others are not protection solutions. But, all five architectures limit the consequences of such attacks by preventing memory regions from being publicly accessible. Moreover, running a privileged OS can be fatal. uVisor does not trust mbed OS, but they are still running with the same privileges. As shown above, an attacker can take advantage of the OS and escalate their privileges.

4.3 Trusted Computing Base

This section compares the TCB size of the studied architectures. It is more appropriate to compare only uVisor and TockOS and ACES because TyTAN and TrustLite are not open source and there is not enough available information about the size of their software components.

uVisor, TockOS and ACES use practically the same MPU (Armv7-MPU). On one hand, uVisor has only 7kLOC (Line of Code), meanwhile TockOS has 11kLOC. However, while uVisor is just a hypervisor and needs an OS to work, uVisor is only compatible with mbed RTOS. mbed RTOS is not trusted, but is executed in privileged mode. uVisor still provides more flexibility than TockOS and is also compatible with three different MPUs (ARMv7m, ARMv8m, and Kinetis). On the other hand, ACES has only 1.3kLOC. This is since ACES is destined to bare metal applications, there is no OS. So, there is no security features like the other architectures. ACES software TCB manages only the MPU reconfigurations and system calls.

4.4 Performance and Memory Consumption

In this section, the performance and memory consumption of uVisor and TockOS are compared, and when possible, ACES and TyTAN too. TyTAN's and TrustLite's codes are not open to the public, so we could not perform our own evaluation tests. For comparison purposes, we used literature data when available. ACES is destined to bare metal applications; therefore, we cannot compare all performance sides, such as process creation. Tests were carried out on an Arm Cortex-M4 board with 2MB of FLASH and 192KB of RAM. uVisor and ACES were already ported to this board, but not TockOS, so the minimum required in order to evaluate and compare the architectures was ported by our own work.

4.4.1 Performances

The evaluation focused especially on some components in the kernel and hypervisor (for uVisor) side. We evaluate mechanisms like task creation, MPU configuration, the context switch, interrupts, and the cost of an MPU fault recovery for uVisor to add an authorized region that was not pre-configured.

Process creation: The process creation scheme differs from one architecture to another. For uVisor with mbed OS, it is done in two steps. In the first step, uVisor loads all boxes' static configuration from the FLASH and allocates memory for every box. The second step occurs during the OS start and consists in creating the corresponding processes. TockOS and TyTAN both support dynamic application loading. Process creation consists of relocating the process, loading the corresponding MPU configuration, and finally, only for TyTAN, the authentication of the process. Table 4.2 presents the performance of creating a process in every architecture. TyTAN has the biggest runtime overhead and this is because TyTAN authenticates every process before loading it, and it takes around 433,400 clock cycles [18] to measure a process. uVisor does not authenticate processes, but it automatically allocates memory for processes. And because of the MPU (Armv7) limitations seen in section 3.1, more processing time is needed to load configuration processes: to round up memory sizes into power of twos, to shift the memory start address to be aligned with its size, and to load the process's access control list. TockOS has the best performance of the

Table 4.2: Performance of creating a process (in clock cycles).

Routine	uVisor	TockOS	TyTAN
Process Creation	5,500	2,900	642,300

Table 4.3: Performance of configuring the MPU (in clock cycles).

Routine	uVisor	TockOS
MPU configuration	390 - 4,450	560

three architectures because it does not authenticate applications. And applications have very small configurations—practically all configurations are static and the same.

MPU configuration: In TockOS and uVisor, the MPU configuration is dynamic; therefore, when a thread-switch or an interrupt occurs, the MPU needs to be configured with the right rules. Table 4.3 shows the cost in clock cycles for such a routine. On one hand, TockOS has only two regions to configure for each application: its text section and its data section, and the configuration is done in 560 cycles. On the other hand, uVisor has two or three static regions, depending on the board, which are the public regions in the SRAM and the FLASH for all applications. The other six regions are filled from the application ACL (Access Control List) with the right rules. Depending on the ACL, the routine can take from 390 to 4,450 cycles. The MPU configuration in TockOS takes less time than in uVisor and this is because uVisor offers more flexibility in defining memory access rules. But, as we will see next, the non-flexibility of TockOS can be very costly when an application needs to access different peripheral registers or other memory regions like the non-volatile SRAM, for example.

Writing in a peripheral register: Table 4.4 presents the results of this test. An application in uVisor and ACES needs less time than in TockOS to write into a register because it was already authorized (in the ACL) to access that memory region. In TockOS, an application can access only its data and text sections, so to write in a peripheral register, the developer needs to define a capsule for the corresponding peripheral. Capsules are part of the kernel and are executed in privileged mode. Calling a capsule from an application requires a Supervisor Call (SVC), then the kernel redirects to the right capsule which writes into the register, and finally, execution mode is changed to user mode and the system returns to the application. It costs around 340 cycles for a simple memory access.

Interrupts: For interrupts handling, the test consists in evaluating the switch in the interrupt

Table 4.4: Performance of peripheral registers writing (in clock cycles).

Routine	uVisor	TockOS	ACES
register writing	6	340	6

Table 4.5: Performance of Interrupts switch in and switch out (in clock cycles).

Routine	uVisor	TockOS
Interrupt	290 - 4,740	3,160

handler and the switch out to resume the execution flow. Table 4.5 shows the results for uVisor and TockOS. For uVisor, it varies from 290 to 4,740 cycles because of the MPU configuration. There is a use case where switching in and out needs only 290 cycles. It is the case when the interrupt occurs during the process owning the interruption handler (See section 3.3). Here uVisor performs ways better than TockOS. In TockOS, it takes 3,160 cycles, because of the MPU configuration. There is also a high latency because the interrupt handler cannot be invoked directly, the kernel has to find the right handler to call.

Context switch: For the context switch, Table 4.6 shows that in ACES it takes 675 cycles to switch from a context to another. TockOS takes 810 cycles, while uVisor takes from 2,040 to 6,100 cycles. This huge difference is, again, due to the MPU configuration. There are up to six more regions that can be configured during this step, and during a process switch, the RPC mechanism needs to drain outgoing RPC queues.

uVisor MPU recovery mechanism: This mechanism shows how a process can configure up to 16 memory regions while the MPU has only eight memory regions. In uVisor two slots are reserved for the public RAM and FLASH, and another slot is reserved for the process heap and stack. So, developers have only five slots to configure. To overcome this limitation, an MPU recovery mechanism allows developers to set rules for more than five memory regions. Then during the MPU configuration, the first five memory regions in the ACL are configured in the MPU. If the application tries to access an authorized address that is not configured in the MPU, the MPU will raise an exception. uVisor retrieves the fault address and checks if it is within an authorized region. Here, uVisor overwrites a memory region with less or the same priority in the MPU and recovers

Table 4.6: Performance of context switching (in clock cycles).

Routine	uVisor	TockOS	ACES
Context switch	2,040 - 6,100	810	675

Table 4.7: Performance of uVisor MPU recovery mechanism (in clock cycles).

Routine	uVisor
MPU recovery	560

Table 4.8: FLASH memory consumption (in kB).

uVisor	TockOS	TyTAN
68.27	72.58	244.08

from the fault. As shown in Table 4.7, this step costs 560 cycles, but it allows an application to have access to more than eight memory regions.

4.4.2 Memory consumption

The memory consumption is divided into two parts. The first part is the memory used when no task is loaded. Table 4.8 shows that uVisor consumes less FLASH than TockOS and TyTAN.

The second part is the memory consumption within the RAM and how uVisor and TockOS manage RAM memory. Table 4.9 presents the initial RAM memory consumption. uVisor alone needs only 2kB, but with mbed OS, 8.60kB are needed. TockOS needs 7kB. The results are very close. The difference between both architectures remains in managing memory for processes. uVisor and TockOS have two different policies. uVisor has a dynamic policy where developers define the amount of memory for a process, then during uVisor boot, it is rounded to match MPU limitations (See section 3.1). In TockOS the amount of memory needed for all processes is the same, and it is fixed within the kernel by the developer. Both policies have their advantages and disadvantages. In uVisor, developers decide if they need a big chunk of memory or just a small one, depending on the application needs. In contrast, in TockOS if there are applications with variable memory block needs, they will have the same amount. This can lead to lots of memory waste, especially when one particular application needs lots of memory space, while others do not need too much. But, in uVisor the fact that memory regions are not fixed can lead to another problem. The ARMv7 MPU limitation can lead to some waste in memory. To limit the waste, uVisor has a routine to order memory regions of applications. These limitations were fixed for the ARMv8 MPU.

Table 4.9: Initial RAM memory consumption (in kB).

uVisor	TockOS
8.57	7

4.5 Discussion

Table 4.10 summarizes some important results of the evaluated MPU-based isolation architectures. All the five architectures have their advantages and disadvantages. Except for uVisor, each architecture is suited to a specific type of applications. uVisor flexibility and small memory consumption make it suited to a large panel of applications. However, the results from Table 4.10 show that the deterministic level can be lost if applications get too complicated. Much effort has been directed towards these architectures, but there is still room for enhancement. For example, in uVisor, root of trust and dynamic loading can be good assets. TockOS is minimalist, and it is designed for tiny and simple embedded systems, such as those with a very limited number of processes and, especially, for applications that spend most of their time in sleep mode. But it does not mean TockOS has less power consumption than other architectures. It has been shown in [66] that TockOS consumes more power than mbed OS, especially when an application requires a lot of computational work, as cryptographic operations do.

For both uVisor and TockOS, runtime performances can be improved. For TockOS, we noted that the MPU was configured at every system tick, even if the OS does not really switch to another thread. Also, during interrupts, sometimes reconfiguring the MPU is unnecessary and there is much time wasted in the kernel finding the right handler. For uVisor, providing developers with up to 16 regions to configure for each process is a good advantage compared to all other architectures. But it has its cost on performance as shown in Section 4.4.1. Maybe the solution for uVisor will be to use a different MPU—one that will allow it to configure all memory regions at boot time. This would suppress the cost of the MPU configuration for every thread-switch and interrupt. But as seen in TrustLite, this might also have a prohibitive cost and might offer fewer memory regions to configure for developers.

TockOS uses Rust in its kernel, which is a memory-safe language. But for low-level operations, TockOS uses an unsafe language. An attacker can control the execution flow from an unsafe code. Applications in TockOS cannot be written in Rust and, to date, they are all written in C/C++. But they are very limited and have access to only two regions. It would be better if they could be written in Rust, too, to lower the attack surface. The three other architectures are written in C and assembly. Unfortunately, only uVisor of the three is open source, so we couldn't review TyTAN and TrustLite code. uVisor is a small hypervisor and its TCB size is around 7kLOC, but it needs mbed OS to work. In Section 4.2, we showed that there is a non-negligible part that runs in the privileged mode. We demonstrated how, from the user mode, we can call the OS and from then execute an arbitrary code in privileged mode. To overcome this, uVisor has to configure the RAM memory as non-executable. Then, to reduce the attack surface, it may need to run the OS with less privilege than uVisor. To do so, we can opt for *para-virtualization*. The timer (*Systick*) used by the RTOS could be handled by a para-virtualized one in non-privileged mode. Using para-virtualized hardware will for sure have an impact on performance, especially on interrupt latency. But determinism can be assured if the para-virtualization is done correctly. We could also reduce the time hogged by the MPU configuration at every thread-switch and every interrupt.

ACES is an automated solution but for bare metal applications only. ACES saves developers from the engineering needed in the other architectures to create isolated execution, allocate the right heaps and stacks, and defining MPU memory regions. However, ACES does not deal with interrupts handlers. These are handled in privileged mode and attackers can take profit of a malicious handler to break system security. Furthermore, ACES can include on average 5% run time overhead [22] comparing to uVisor. This because ACES offer a finer compartmentalization granularity and its switch context is quite costly. Although, this fine granularity reduces ROP attacks on average by 94.3% [22].

Another issue in all these solutions is IO peripherals protection. MPUs in the five architectures are interfaced with the CPU, which means only software memory accesses are controlled. Because IO peripherals are outside the CPU, they can access all memory regions bypassing all architectures' security measures. Overcoming IO peripherals attacks is very challenging, especially from a software standpoint. TockOS kernel has a software abstraction layer that partially limits the MPU circumvent. Rust memory safety ensures the kernel exposes memory-mapped registers safely, so applications cannot write arbitrary values beyond the boundaries [54]. A complete software solution may be difficult and costly in terms of performance. A software/hardware or just a hardware solution may be better.

Table 4.10: Summary of the evaluated MPU-based isolation architectures.

	Security features							Memory consumption				Performance (clock cycles)					
	Inter-Process Communication	Root of Trust	Dynamic loading	Exception Handling	Application reboot	DMA protection	Memory safety	MPU Access Control List	TCB size (kLOC)	FLASH Consumption (kB)	RAM consumption (kB)	Process creation	MPU configuration	Register writing	Interrupts	Context switch	MPU recovery
uVisor	x	o	o	x	o	o	o	16 per process	7	68.27	8.57	5,500	390 4,450	6	290 4,740	2,040 6,100	560
TockOS	o	o	x	x	x	xo	xo	2 per process	11	72.58	7	2,900	560	340	3,160	810	o
TyTAN	x	x	x	x	o	o	o	18	?	244.08	?	642,300	?	?	?	?	o
TrustLite	x	x	o	x	o	o	o	32	?	?	?	?	?	?	?	?	o
ACES	o	o	o	x	o	o	o	8	1.3	-	-	-	-	6	?	675	o

x: Yes, o: No, xo: Partial, ?: NA, -: Non-relevant

4.6 Summary

In this chapter, we presented our study of most known MPU-based isolation architectures. The study covered a comparison of isolation robustness, and security features, and performances and memory consumption of some architectures, because unfortunately not all architectures are open.

This study shows that all architectures provide strong isolation, and they limit the consequences of software attacks, and make it harder for an attacker to reach their goal. We evaluated the performance costs of MPU-based isolations, the results of costs were consistent with the policies chosen in each architecture. Nevertheless, the study presented some weaknesses and limitations in every architecture and demonstrated how reusing a gadget from the OS can be fatal.

Finally, we presented some ideas of possible works for a better isolation and to reduce the runtime overhead. In the next chapter, we present our architecture, Toubkal, that was designed based on the outcome of the preceding two chapters.

Chapter 5

Toubkal: Hardware-based Isolation and Attestation Architecture

This chapter presents the design of Toubkal architecture. We detail how we achieve the targeted levels of protection in the context of our threat model (section 2.2.3). In chapters 4, we studied in detail five lightweight architectures. We reviewed and compared different aspects to understand what is missing to offer a highly secure architecture. The outcome of this study (section 4.5) leads us to design a hybrid architecture to enforce lightweight devices security. In this chapter, we present Toubkal design to provide isolation and attestation for lightweight devices. Figure 5-1 gives an overview of the architecture.

Toubkal is composed of two hardware modules to control memory accesses on different levels. The first hardware module consists of controlling memory accesses of the various peripherals with memory access. It catches all the signals going to the memories and checks the permission accesses configuration to allow or not the memory access.

The second hardware component consists of controlling the memory access to the most critical software, which is the security monitor. The main goal is to create a separation between the latter and the rest of the software. It protects its (security monitor) data and controls all the calls to its code.

Toubkal is also composed of the security monitor, the software component. This component is composed of two parts: the root, which is non-changeable, and the monitor. This component is

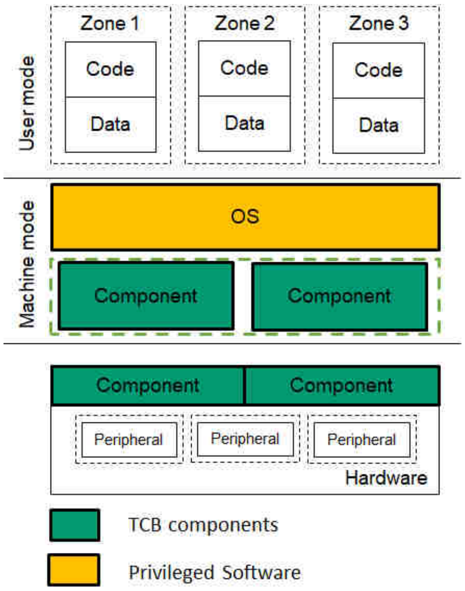

Zone 1
Zone 2
Zone 3
Code
Data
User mode
Machine mode
OS
Component
Peripheral
Hardware
TCB components
Privileged Software

Figure 5-1: Effective isolation and attestation architecture overview

responsible for configuring the two hardware components to make the architecture more flexible and is also responsible for establishing trust in any data or code. It offers an attestation service to verify their authenticity and integrity.

All these components form the Trusted Computing Base (TCB) of our architecture. They are modular and can work independently. However, excluding a component results in reducing the protection desired in this thesis. First, we will start by presenting an overview of Toubkal architecture. Then, we will present, in-depth, each component we designed and developed.

5.1 Toubkal Overview

This section presents an overview of Toubkal design. The primary contribution of this chapter is a hardware/software co-design that addresses the limitations cited previously. Toubkal combines two hardware modules and a software component.

5.1.1 Toubkal Architecture

Toubkal is mainly composed of three components; two hardware components and a software component (see Figure 5-2): *the master memory protection (MMP)*, *the execution aware protection (EAP)*, and *the security monitor (SM)*.

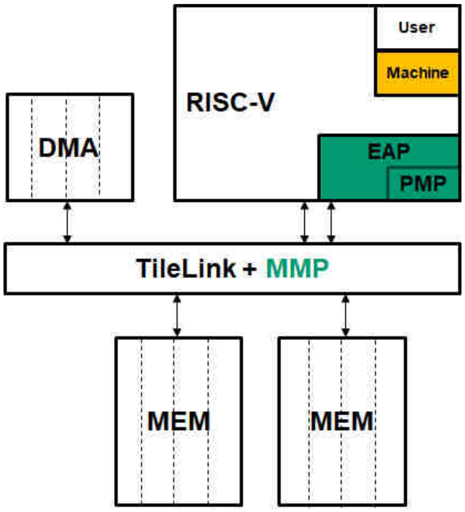


Figure 5-2: Major Components of Toubkal: It is a hardware/software co-design composed of three main components; the MMP which is the hardware module control memory access of the different peripherals, like CPUs, DMA, etc. The EAP which is the hardware module to protect and create a strong separation of the security monitor from all the rest. Finally, the security monitor which is the only software responsible of enforcing system security. The PMP is optional and can be used to create multiple zones for the user mode.

The master memory protection (MMP) is responsible for controlling IO peripherals' memory accesses. The MMP contains configurations of memory regions and their access permission and it checks memory accesses to authorize or not a peripheral from accessing a specific memory address. The MMP has a master look-aside buffer (MLB), where it stores memory regions configurations. It is integrated within the interconnect bus to control memory accesses from all the peripherals to memories. The MMP supports multiple policies and options to help designers generate the right hardware module.

The execution aware protection (EAP) is responsible for creating isolation around the security monitor. As there are only two execution modes (the user mode and the privileged one), the security monitor and privileged code (such as the OS) can share the same privilege. This results in a big attack surface. Therefore, the EAP main goal is to separate the security monitor with the rest, and to control every access of the security monitor's code and data. The same way and for the same reasons, we can have another separated zone (zone 3 in figure 5-3), called D-zones (D for Detached).

The security monitor (SM) is a software component composed of two parts: a tiny and immutable root and a changeable monitor. The root is the only trusted software because it is immutable. It is responsible for configuring the MMP and the EAP, and for verifying the monitor authenticity and integrity with a hardcoded key. The security monitor is kept relatively small to make verification easy.

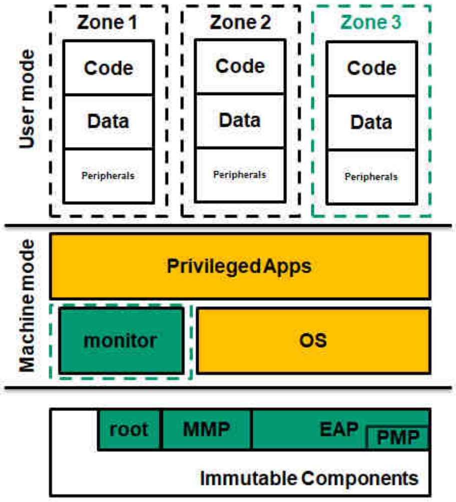

User mode
Zone 1
Code
Data
Peripherals
Zone 2
Code
Data
Peripherals
Zone 3
Code
Data
Peripherals
Machine mode
Privileged Apps
monitor
OS
root
MMP
EAP
PMP
Immutable Components

Figure 5-3: Toubkal's architecture

Figure 5-3 presents an overview of Toubkal's architecture. An important goal of Toubkal is to offer isolation on multiple levels. In most lightweight devices, there are only two modes; user mode and machine mode, which the most privileged mode in RISC-V architecture (similar to the supervisor mode in the Arm cortex-M architecture). On the user mode level, we can load multiple applications into separated zones. All zones are equally untrusted. Yet we can differentiate between two types of zones. we have normal zones that can be processes and managed by an OS, and D-zones that are completely separated from the user and the privileged code. Normal zones can be created using the Physical Memory Protection (PMP). Inside D-zones, we can also have multiple separated zones using the same PMP.

D-zones are useful to embed sensitive libraries or applications that do not need an OS. As we showed before, OSes are not trusted, therefore, D-zones are useful to embed applications and libraries we want to protect from OSes' weaknesses. The EAP module helps to protect them from unauthorized accesses.

On the machine mode, we want a clear separation between the security monitor and the other privileged code to reduce the attack surface of the security monitor and avoid security issues like the one showed in mbedOS+uVisor in 4.2. The EAP needs to control every access to the security monitor and ensure that there is no leak of the latter's secrets.

Both D-Zones and the security monitor have entry points. Any jump inside these zones without passing by the entry points needs to be stopped. Calling these protected zones has to follow a defined set of steps. The EAP needs to control the transitions from a step to another and stop any non-authorized transition.

Toubkal hardware modules are in Chisel3 [13] and the security monitor is in C language. It is compatible with the existing RISC-V based Instruction Set Architecture (ISA). The MMP is also compatible with Arm based platform as it does not require any change to the main core.

5.1.2 Modularity

A major point and challenge of Toubkal was to keep it modular. This means that each component can work independently from the others. Consequently, Toubkal can work with the MMP only, with the EAP only, or with both. In sections 5.2.1 and 5.2.2 we present the MMP and the EAP as they are working outside Toubkal ecosystem.

Although modularity may seem interesting, it impacts the level of security. For example, if a platform integrates the MMP alone, it will have some drawback in security. The primary goal of the MMP is to protect from IO peripherals non-authorized accesses to memory. The MMP needs to be configured from a software component. This software is vulnerable to privilege escalate attacks. If an attacker succeeds in escalating the privilege, they can change some configurations. The EAP adds another layer of isolation to isolate the security monitor, the only software able to change the MMP configurations.

Same for the EAP, if a platform integrates it alone, the system will be vulnerable to IO peripherals attacks and an attacker can access sensitive data processed by the security monitor.

Putting the MMP and the EAP all together with the security monitor makes the platform more secure with multiple layers of protection.

5.2 Toubkal Design in Detail

In this section, we provide a detailed description of the different hardware and software components we designed.

5.2.1 The Master Memory Protection

This section presents the design description and evaluation of the Master Memory Protection (MMP). It is a modular system that offers strong separation of different hardware modules within a system. The MMP has been designed in such a way that it can easily be adapted to the system needs in terms of security, safety and performances. It does not require any change in the existing hardware modules. Below, we present a detailed description of this block.

The major challenge of the MMP was to develop a flexible design and to keep it small with a negligible run-time overhead. This section starts with describing our initial analysis of the design, then presents our proof-of-concept implementation.

MMP Major Components

The MMP is composed mainly of the Master Look-aside Buffer (MLB) the uCode block, in addition to some logic for the look-up and match, security checks and MMP configuration (see figure 5-4). In this work, the MMP is interfaced with the AHB but it can be positioned in any other position that catches communication between Masters and memories. The abstraction layer aims to facilitate the integration of the MMP in different systems and positions.

The MMP uCode is responsible of memory mapping all MMP registers and configuring the MMP. uCode can be configured from the security monitor. The MLB is a set of registers to store regions configurations or look-up&match some addresses. This section presents and discusses two different implementations of the MLB, the Unique MLB and the shared MLB. uCode is connected to MLB registers to store regions configurations. And then, there is the abstraction layer. This layer is customizable and aims to facilitate the integration of MMP in different systems and spots.

Master Look-aside Buffer

The MLB is designed to store memory regions configurations, it looks-up&matches addresses. To store configurations, the MMP uses registers, which are costly, but guarantee rapid access. The

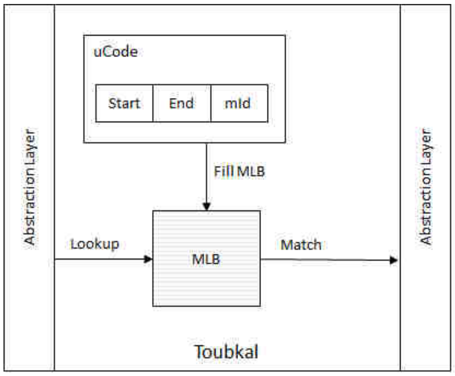
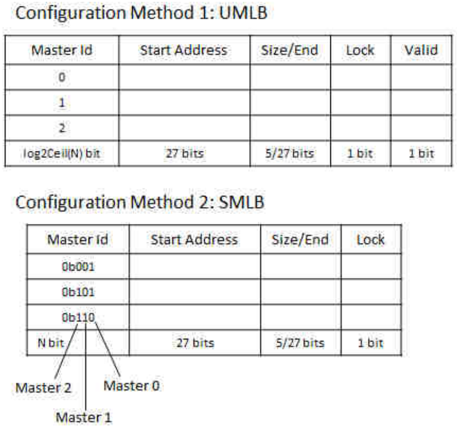


Figure 5-4: Major components of hardware block of the MMP: The uCode module contains information to recognize the security monitor which is responsible of configuring the MMP. This module is linked to the MLB. The MLB is a set of registers to store regions configurations. The abstraction layer is customizable and aims to facilitate the integration of the MMP in different systems and at different spots.

MMP proposes different ways to store configurations. As shown in figure 5-5, designers can choose the number of memory region slots, the granularity, the address width, the number of concerned Masters and the method to attribute slots to Masters.



Configuration Method 1: UMLB

Master Id	Start Address	Size/End	Lock	Valid
0				
1				
2				
log2Ceil(N) bit	27 bits	5/27 bits	1 bit	1 bit

Configuration Method 2: SMLB

Master Id	Start Address	Size/End	Lock
0b001			
0b101			
0b110			
N bit	27 bits	5/27 bits	1 bit

Master 2
Master 0
Master 1

Figure 5-5: MLB slots: The figure presents the different ways to store configurations in the MLB for a 32 bit address space. N represents the number of masters. The first method, called UMLB (U for Unique) stores one configuration per Master per slot, while the second method, called SMLB (S for Shared), stores one configuration per slot but for multiple masters. In the SMLB, there is no need for a valid bit as each Master has a corresponding bit in the first column from the left. The value 0 means the configuration is not valid for that master and vice versa. The third column from the left refers to size in case of coarse-grained granularity, and to end address in case of fine-grained one.

The first important decision to make in the MMP is to choose between configuring memory regions **white-list** or memory regions **black-list**. This choice is strategic and depends on which case would lead to fewer slots to configure. We added this option as we believe for isolation at this level, contrary to MPUs and MMUs, it is more interesting to create a black-list rather than a white-list. It is more likely to prohibit a memory region rather than allowing it. Indeed, the way of thinking at this level is different compared to the application level. Thus, in the rest of the chapter we will be more talking about prohibiting memory regions to Masters rather than allowing them.

The MMP offers two ways to organize slots (see figure 5-5). The first method is called Unique MLB (*UMLB*) because each slot is unique to a Master, and the second method is called Shared MLB (*SMLB*) because a slot can be shared between multiple Masters. The *UMLB* consists in storing in each slot a memory region configuration for a specific Master. While the *SMLB* consists in storing in each slot a memory region configuration for multiple Masters. Each way has its advantages and its drawbacks, and each is more suitable to specific use cases. While the *UMLB* requires less logic to match or insert the Master Id, the *SMLB* can require fewer slots (therefore fewer cells area) in case some Masters share some common permissions. In the SMLB, there is a bit for each Master. When the bit is up, it means that the configuration is valid for the corresponding Master and vice-versa.

Concerning the number of slots, the MMP accepts up to 16 slots. Because of the high cost of registers, we limited the number of slots to 16 slots. Designers can choose between 4, 8, 12 or 16 slots. The size of the MMP grows proportionally to this number. Section 6.2.1 shows how the area is impacted by the number of slots.

The other parameter is the granularity of the MMP. This work proposes two granularities. For fine-grained granularity, the MMP proposes regions multiple to 32 bytes and the region address start is aligned to 32 bytes. And for coarse-grained granularity, the MMP proposes regions power of two and the region address start is aligned to its size. The second granularity is more constraining in comparison to the first one. However, as figure 5-6 shows, the coarse-grained one requires less logic than the fine-grained one to match addresses.

To configure memory regions in the MMP, MLB registers can be hard-coded for more resiliency and less flexibility, or memory mapped and then the uCode is responsible for configuring them during software execution for more flexibility and dynamism. Making the MMP accessible from the running software adds its attack surface. We discuss this point in the next section.

The MMP Policies

We put in place different policies to generate different designs of the MMP. These choices impact the hardware area cost. Section 6.2.1 compares all the options. We implement three policies, *static*,

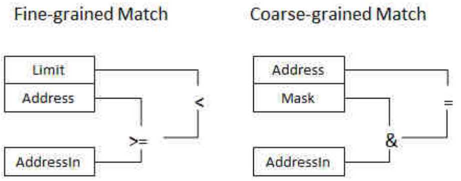


Figure 5-6: A fine grained match is more complex than a coarse grained match. The second one compares the address with the base address and the limit address, while the first one compares the logical combination of the address and the mask corresponding to the size of the region with the base address.

semi-static and *dynamic*.

The *static* policy is a security policy that focuses on resiliency rather than flexibility. In this policy, memory regions configurations are hard-coded, or loaded using the Read Only Memory (ROM) or a One Time Programmable (OTP).

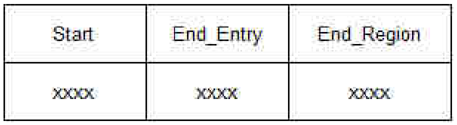
The *semi-static* policy is a little more flexible than the first one. Like the *static* policy, some configurations can be fixed from the beginning, but it offers the possibility to add other configurations during execution. To do so, we add a uCode that has to be run in privileged mode and we memory map some registers. These registers are only accessible from privileged mode. The uCode must be a predefined memory region, which means that outside this region MMP will not configure itself. Once a configuration is added, it cannot be modified or removed, although, this policy works only with the SMLB configuration, therefore, it is possible to validate or invalidate the configuration for each Master by flipping the corresponding bit. This can be useful in the use case presented in section 2.3.2. It is also possible to lock a configuration so no one can change it until system reset.

The *dynamic* policy is the most flexible but the less secure of the three policies. From the uCode, it is possible to add, remove or change all configurations. The uCode runs under the same condition as in the *semi-static* policy. There is also a lock bit to lock a configuration so it cannot be modified until system reset.

In the next chapter, we evaluate The MMP, compare and discuss the differences between each policy in terms of performances, cells area and security.

5.2.2 The Execution Aware Protection

This section presents the EAP module, which is mainly used, but not limited, to create a strong separation of the security monitor from other software components. The EAP was designed to answer some security issues (e.g. running an untrusted OS and a security monitor with the same privilege) we found in lightweight isolation and attestation architectures like [9, 18, 22, 48, 53].



Start	End_Entry	End_Region
xxxx	xxxx	xxxx

Figure 5-7: Registers to configure the protected text sections: Regions cannot overlap. Entry points to the protected region must be inside the start address and End_Entry. Entry points must be as few as possible to reduce attack surface.

The Execution Aware Protection (EAP) primary objective is to guarantee a strong separation of the security monitor from all the rest of the system. It can also be combined with the physical memory protection (PMP) to create the D-zones (D for detached zones), which are zones running in user mode and are protected from the privileged mode. EAP offers hybrid isolation, in other words, isolation is implemented by checking the instruction address, the context configuration, and the execution mode, to allow or not a data memory access. Figure 5-7 shows registers to configure code sections and entries sections for both the security monitor and D-Zones. In this work, Entries section and code section have to be a continuous space.

The EAP checks all instructions to control the access to the security monitor and D-zones. It controls three text memory regions; the root, the monitor, and the D-zones. The EAP requires entry points for each one. The primary goal of entry points is to force the program to call these regions from the defined entries. This way, the EAP prevents attackers from randomly calling sensitive functions to extract sensitive data from the protected regions.

Memory distribution and access control

The EAP divides the memory into four main groups. Each group has a sensitivity priority. At level zero, which is the most sensitive, there is the memory belonging to the security monitor. At level one, there is the memory belonging to D-zones. At level two, there is memory belonging to the privileged world. And finally, at level three, which is the least sensitive, there is memory belonging to the user world.

Levels zero and two are running in privileged mode, and levels one and three are running in user mode. For convenience and performance matters, code within level zero has access to all other levels. And code within level two has access to level three. Concerning level zero, the reason

Table 5.1: Notations used in designing and verifying the state machine of EAP

Notation	**Description**
PC	Program Counter
RC	Root Code
REC	Root Entry Code
MC	Monitor Code
MEC	Monitor Entry Code
DZC	D-zones Code
DZE	D-zones Entry Code
SMM	Security Monitor Data Memory
DZM	D-zones Data Memory
KM	Memory area for key storage
sRomEnt	State when PC is in the Root code entry point
sRomIn	State when PC is legitimately inside the root
sMonitorEnt	State when PC is in the monitor entry points
sMonitorIn	State when PC is legitimately inside the monitor
sDzoneEnt	State when PC is in the D-zones entry points
sDzoneIn	State when PC is legitimately inside the D-zones
sNone	State when PC is in the rest of code
sKill	State when there is a non-authorized access
oldState	The state just before we check the new PC
newState	The state after checking the transition and memory access
Daddr	Data address for memory access

Table 5.2: Memory access control for all rings

From/To	level 0	level 1	level 2	level 3
level 0	rwx	rw	rw	rw
level 1	-	rwx	-	-
level 2	-	-	rwx	rw
level 3	-	-	-	rwx

behind this choice is obvious; it is part of the TCB and is the software part enforcing the device security. Concerning level two, the reason behind it is justified by the fact that level two is running in privileged mode and might embed an OS. An OS that is managing processes running at level three. Table 5.2 summarizes the access control list between levels.

Level zero and one are protected from other levels using a program counter based isolation. Level zero is protected from level one using a program counter based isolation, and level two is protected from level three using execution mode and the PMP based isolation. And finally, the EAP can use the PMP to create multiple separated zones inside levels one and three.

Concerning level zero and level one, which use a program counter based isolation, we define entry points for each level. The entry points are used to force the outside world (level two and level three) to call available services/functions from the defined entry points. Therefore, the entry points are the only gates to level zero and level one. Doing so prevents ROP attacks by reusing and combining gadgets from level zero and level one to leak secrets and change the system's behaviour. However, it does not prevent an attacker from diverting the system behaviour if an attacker exploits a bug inside the code level. Entry points prevent only ROP attacks being mounted from outside the level.

EAP design allows level zero and level one to have multiple entry points. While this point can be convenient and offer flexibility, it is recommended to have as few as possible. Entry points can be considered attack's entries. Therefore, to reduce the attack surface, we need to have few entry points. Entry points are the only codes that can be executed from outside.

Each level has text and data sections. Currently, the EAP only supports continuous text and data sections for the security monitor and D-zones (levels zero and one). The rest of the code can be loaded anywhere else in the available memory. Furthermore, level zero and level one cannot overlap with each other or with the rest. While levels two and three can overlap with each other (the case of an OS and its threads).

The EAP also provides the possibility to divide levels one and three into multiple separated zones. This time, the EAP uses the PMP to create such separation. This is the **third layer of isolation**. Each zone has its text and data section. We do not go in detail at this level of isolation. However, MultiZone [14] can work perfectly in our architecture to provide this layer of isolation.

The EAP Execution Flow

Here, we will explain how it works inside the EAP module. The EAP uses the program counter based isolation to check access rights for every instruction. For this reason, our protection mechanism must be instant, which is the case. The EAP uses registers that are access-fast to store each security monitor and D-zones configurations. Moreover, the circuit is only combinatorial. Therefore, no extra cycle is needed to check for access rights.

There are three essential points in the EAP that we will tackle in this part: First, the EAP has to ensure that the system starts from the root so it (root) can establish trust in the whole running software. Second, as it has been said, the EAP creates four levels of protection. The program flows from one level to another. Therefore the EAP has to ensure that transitions are secure and safe. Third and finally, the EAP is responsible for checking memory accesses of the running program.

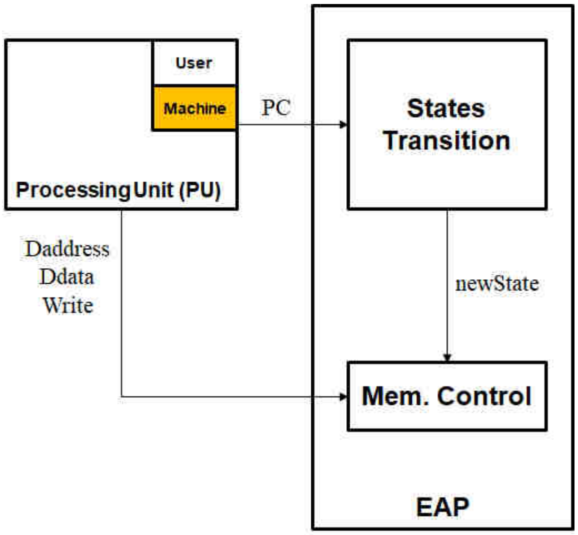


Figure 5-8: The EAP is mainly composed of two components. The *state transitions* component is responsible of checking if the transition from oldState to newState is authorized. Then, the second component, *memory control*, is responsible of checking the memory access data if it is authorized from that state.

In the EAP, there are nine different states: *sInit*, *sRomEnt*, *sRomIn*, *sMonitorEnt*, *sMonitorIn*, *sDzoneEnt*, *sDzoneIn*, *sNone*, and *sKill*. The states of level zero are sRomIn and sMonitorIn. sDzoneIn is the state corresponding to level one. sNone is the state corresponding to levels two and three. At system start/reset, the EAP is in the state sInit. Then, there are sRomEnt, sMonitorEnt, and sDzoneEnt, states of the different bridges between the other states. Finally, sKill is the state reached when there is a non-authorized action. Figure 5-8 presents the two main components of the EAP. The *state transitions* one, and the *memory control* one. The *state transitions* module is the one responsible for controlling the validity of each transition from a state to another. This module requires the EAP configuration, the EAP current state, and the PC address. This component computes the attempted future state, and then check if the transition is authorized or not. In case the transition is allowed, the *memory control* takes the hand. This component requires mainly the data address, the EAP configuration, and the read/write register value. It checks if the current state can access the data address. In the case of the security monitor, the EAP sends a signal to the PMP to bypass the latter memory access control as the security monitor has access to all memories.

1) System Initialization: At system start/reset, the EAP is in the state sInit. Normally, the root should be the first program to execute, and it has to start from its entry point. The EAP ensures that, at system initialization, the first code called is the root code entry point, and then comes the root. The root entry point can be called only after the state sInit. The root services are not callable from the outside. Only the monitor is able to jump into the root code.

2) States Transitions: Transitions from one state to another must follow the defined policy. This means that we define authorized and non-authorized transitions to enforce the isolation. The EAP always checks if transitions respect the rules, which are illustrated in Figure 5-9. The black arrows are the allowed transitions. If a program tries to follow another path, it transits to sKill state and resets the system.

3) Memory Access Control: After the EAP checks the correctness of the transition, it checks the data read/write memory access. Each level is attributed a data memory area. The EAP checks if the current state has access to the requested data. Table 5.2 presents the access control list. Except for level zero, which has full access, the other levels can just read/write their data or code.

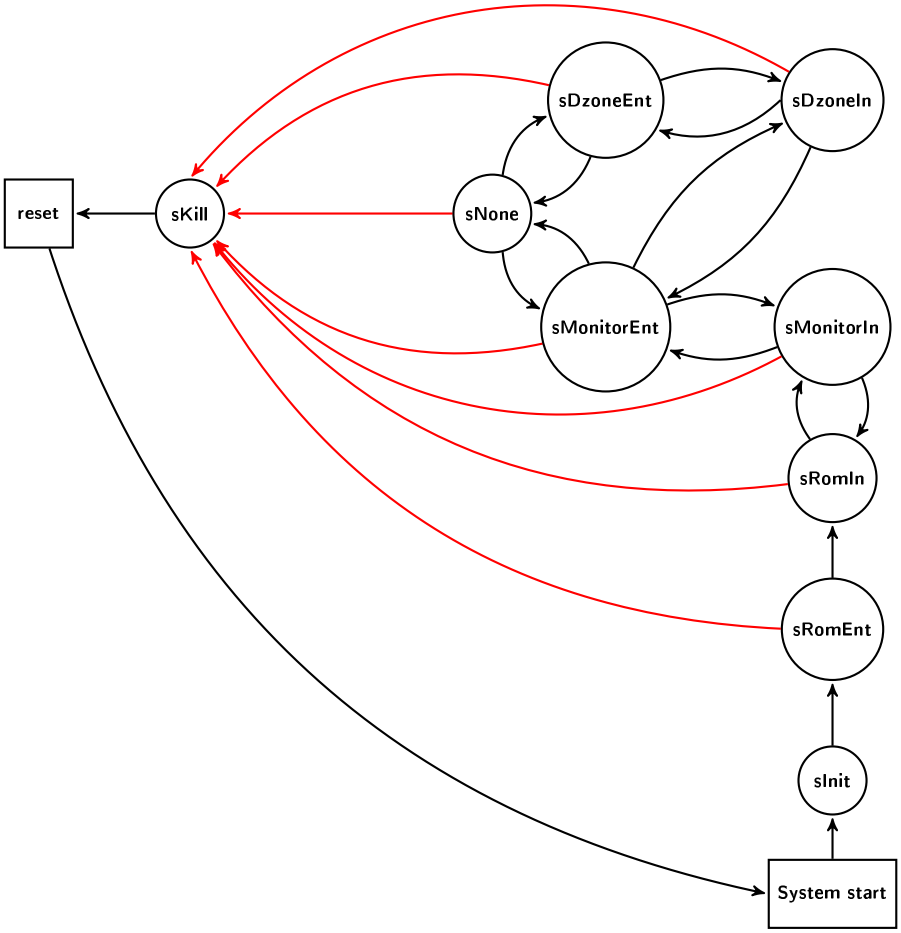

reset
sKill
sNone
sDzoneEnt
sDzoneIn
sMonitorEnt
sMonitorIn
sRomIn
sRomEnt
sInit
System start

Figure 5-9: States Transitions: Black arrows represent authorized transitions. Any other attempt will lead to a transition to sKill state as illustrated in red arrows. When the EAP reaches the state sKill, it resets the system.

5.2.3 The Immutable root

The security monitor is the software component of the TCB. It runs at the highest privilege and has access to all memories. The security is isolated from all other components, software and hardware, by the EAP and MMP. It is composed of two parts; *root* and *monitor*. This section presents the immutable *root* and how it achieves the attestation of the integrity and authenticity of the untrusted software.

The root is immutable because it is stored in the ROM. At boot time, it is the only trusted

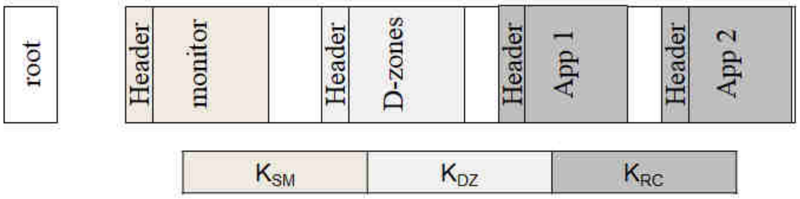


Figure 5-10: This figure presents the keys used to compute the hash to verify the integrity of each application.

software in the architecture. Its main function is to verify the integrity and authenticity of the monitor. The root provides the HMAC algorithm to compute a cryptographic hash of the monitor using a hard-coded Key K_{SM}. Using the hash, K_{SM} and a key derivation function `genKey()`, the security monitor generates a cryptographic key K_{DZ} that will be used to verify D-zones code. The same process is repeated to generate a third key K_{RC} to verify all other applications. Figure 5-10 presents an illustration of the different updatable applications embedded within the system and the respective keys used to verify the authenticity and integrity of their code.

Software attestation

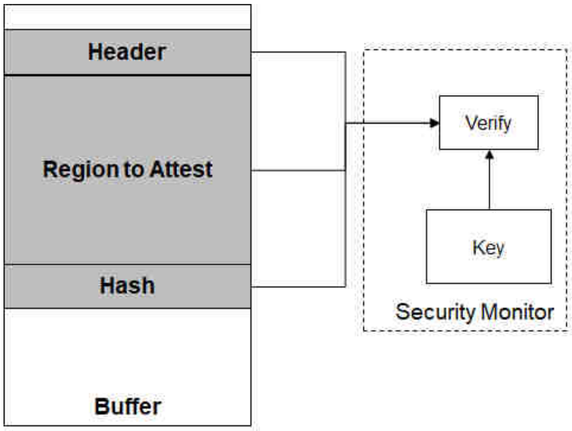


Figure 5-11: Software Attestation consists of retrieving memory region information from the header, computing memory region hash, and comparing the computed hash with the provided one.

The root offers the possibility to attest any piece of code or data. The root, as shown in Figure 5-11, retrieves from the header the start address, the size of the code, the key, and the pre-computed HMAC. The root, with the right key, computes the HMAC and then compares it to the given HMAC. Concerning the HMAC calculation, the root uses HACL* library [105]. The

HACL* library offers an HMAC-SHA256 implementation that is verified. HACL* was implemented and verified using F*. Then, The C version of the library can be generated.

The MMP and EAP protect the root and its data from being misused or leaked. The EAP checks if transitions were respected, and protects its data from being accessed from malicious software, while the MMP protects it from being leaked using an IO Peripheral.

The first step in the root is gathering information about the monitor. Information like the monitor's location in memory, its size, and the location of the pre-computed HMAC. Before using this information to attest the monitor, the root attests the integrity of this data using the same HMAC algorithm with the key K_{SM}.

Once the root checks the monitor integrity and authenticity, it generates the two symmetric keys K_{DZ} and K_{RC}. First, the derivation function `genKey()` generates K_{DZ} from the monitor hash and K_{SM}. Then, it generates K_{RC} using the same hash and K_{DZ}.

To avoid generating the keys every system start/reset, the computed monitor hash is stocked alongside the K_{DZ} and K_{RC}. Once the monitor hash is computed and verified, the root checks if the monitor hash has been changed. If not, the generation key step is skipped.

The idea of generating other keys for attestation and encryption is to reduce and limit the use of K_{SM}. K_{SM} is hard-coded. Therefore it cannot be changed in case it is leaked. By limiting the use of this key to the root only, we reduce the chances of being leaked. K_{SM} is stored inside the EAP, and the root uses specific memory-mapped registers to retrieve it, use it, then clean it from memory.

In this work, interrupts are disabled during security monitor run-time. The EAP does not support secure interrupts management. However, it can be supported using a software component that switches-in and switches-out in a secure way. Although this method, if well implemented, offers a good security level, it can introduce a lot of run-time overhead. In this work, if an attacker, for example, tries to run Time-of-check-to-time-of-use (TOCTTOU) attacks, where they program a timer to occur every t time to retrieve data while the security monitor is computing the hash, the EAP will block the access as the transition to the interrupt handler would not be respected.

5.3 Summary

In this chapter, we presented the design overview of Toubkal. It provides multi-layer isolation to create equally untrusted zones and enforce lightweight devices protection. The first layer of isolation is assured using the Master Memory Protection (MMP). The MMP controls IO peripherals' memory accesses. It reduces the memory range for each IO peripheral. This way, attackers cannot use these peripherals to access sensitive data or inject malicious code.

The second layer of isolation is provided by the Execution Aware Protection (EAP). The primary goal of this module is to provide strong isolation of the security monitor from all the rest; other applications and IO peripherals. The EAP divides the memory into four separated levels. It mixes multiple methods to establish a fast and flexible separation with a good level of security. Entry points can be defined to ensure critical applications are called from the defined entries. The EAP ensures that all non-authorized jumps are blocked.

Finally, we presented the immutable part of the security monitor called *root*. This software component is stored in the ROM, and it is the only software that is indeed trusted. The EAP ensures that the root is the first software called. The root's primary goal is to establish trust in the monitor by verifying its integrity and authenticity.

In the next chapter, we analyse the security of Toubkal and evaluate its silicium area and performance.

Chapter 6

Toubkal Security Analysis and Evaluation

This chapter presents a detailed security analysis of Toubkal. The security analysis consists in defining security properties that we prove using formal verification to validate the correctness of the design protection. Then, we present the evaluation results of the hardware area and performance of attestation and key generation.

6.1 Toubkal Security Analysis and Validation

This section consists of defining the security properties required for strong isolation and attestation. Then, we verify these properties with the computer-aided formal verification.

The computer-aided formal verification consists basically of using mathematical reasoning to prove that the design meets all the specifications [46]. It is composed of three steps. The first one consists in modeling the system in a formal model. The second step consists of specifying the properties that the design must verify. Finally, the third step consists of checking the formal model against the desired properties.

Here, the computer-aided formal verification focuses on the Execution Aware Protection (EAP) module as it contains multiple states and controls the whole software. The Memory Master Protection's (MMP) design is very simplistic and has only two states, normal state, and micro-code state.

However, we will use the protection layer provided by MMP to prove some security properties.

We model* the EAP as a Finite State Machine (FSM). We provide verification and proof of the isolation security provided by the EAP by proving the security properties defined below. There are nine states in the EAP: sInit, sRomEnt, sRomIn, sMonitorEnt, sMonitorIn, sDzoneEnt, sDzoneIn, sNone, sKill. Table 5.1 presents the different notations used here. We are using NuSMV [21] to draw our formal model and the Linear Temporal Logic (LTL) specifications to define our isolation and security properties.

The NuSMV is an extension of the symbolic model checker [62]. It has been designed as a structured and open architecture and is widely used by industrials because it is close to their standards. NuSMV is used to model both hardware and software components. First, we model the components as FSMs. Then, we express formally the properties the model has to satisfy. Finally, we check that the properties are really respected and there is no counter example.

LTL specification is evaluated over infinite sequences of states. Other than the propositional calculus, such as conjunction "&" that we use to prove our model, the LTL specification provides temporal connectives, such as:

- **Future "F"**: F p is true if p is true at a future state.
- **Global "G"**: G p is true if p is always true.
- **neXt "X"**: X p is true if p is true at the next state.
- **Until "U"**: p U q is true if q is true and p was true at all the states before.

The propositional calculus with the temporal connectives allow to create sufficient rules to prove the model security soundness. All the security properties defined in 6.1.1 are specified in LTL format and verified using NuSMV. The latter checks all reachable states and verify if the properties are respected. Below, we discuss how these properties prove the security and safety of our architecture.

6.1.1 Security Properties

We define the security properties required to guarantee a good level of protection. The security properties concern both isolation and attestation. Then we verify our design and prove that it satisfies these properties. We can divide them into two categories, **execution isolation**, and **assets protection and confidentiality**.

---

* The model can be found in https://bitbucket.org/halazouna/model-checker-eap

Execution Isolation:

- **SP1 States sequence correctness** The design must respect states order to be sure it is run safely and securely.
- **SP2 Immutability:** root code has to be immutable, otherwise, an attacker can change it to a malicious code and break the system security. The monitor and D-zones codes have to be semi-immutable. This means the code cannot be changed and run directly. If we want to update those codes, we have to call the root to check their integrity before calling them.
- **SP3 Controlled Call:** Security Monitor and D-zones must be called from the defined entries.
- **SP4 Interrupt Handling:** Interrupts can present a high risk to mitigate EAP security. We need to make sure that all interrupts are handled safely.
- **SP5 Configuration:** EAP configuration must be possible only from the root, and it is done once and cannot be changed until system reset.

Assets Protection and Confidentiality:

- **SP6 Code confidentiality** Security monitor and D-zones codes are kept confidential from reads attempt from the outside.
- **SP7 Key confidentiality** The hard-coded Key can only be accessed from the root. This limits the access scope. The other keys are accessible from the whole security monitor.
- **SP8 Secrets confidentiality** For each, security monitor and D-zones, there are stacks and heaps that are reserved respectively. Accesses to these regions are only possible from inside each region respectively.

6.1.2 Execution Isolation Computer-Aided Verification

For each property, we provide the formalized version in LTL specification and use NuSMV to check the soundness of the property. Some security properties are translated into multiple LTL specifications.

State Sequence Correctness

There is a sequence of states to be respected; any tentative to skip a state must lead to an error. The challenging point here is to find a balance between the ease-of-use for developers, the low run-time overhead while guaranteeing a good level of security. In this part, we will study and verify all the possible, authorized and non-authorized, transitions from the following states: sRomIn, sMonitorIn, sDzoneIn, and sNone. Then, we will prove that the EAP guarantees safe transitions only using LTL specifications. We present the cases of the sRomIn (see Figure 6-1) and sMonitorIn (see Figure 6-2) states. As for the states sDzonesIn and sNone, we proceed the same way.

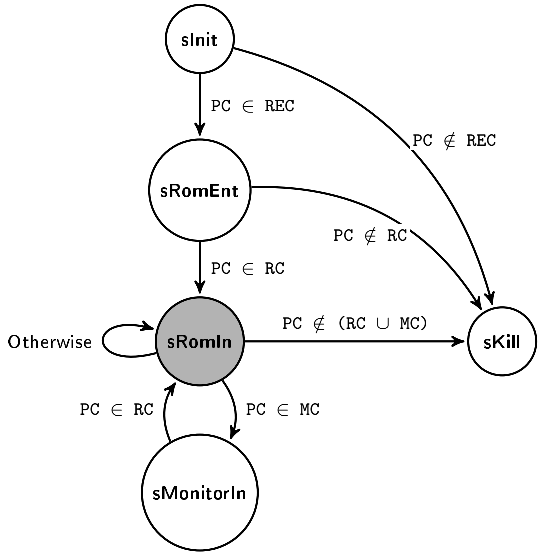


Figure 6-1: State sequence correctness around the state sRomIn

Root execution verification: The root is part of the security monitor, and it is the first software to be called. Figure 6-1 shows the different state that are directly connected to sRomIn, the state where the system is running the root code. When the device starts, the EAP state is in sInit. The device has to boot from the root entry point (sRomEnt). The root has only one entry point from outside the security monitor and can be called only during boot time. When the root finishes its workload at boot time, it gives the hand to the monitor. In this case, there is no need to call the monitor entry points to call it. The same happens in the other way, the monitor is the only one able to call root services at runtime.

The next three LTL specifications prove the correctness of state sequences around the state sRomIn. During the transition, oldState refers to the EAP state just before the transition attempt, and newState refers to the EAP future state.

`LTLSPEC G(oldState=sInit)&(PC∉REC) -> X(newState=sKill)` (a)

`LTLSPEC G(oldState=sRomEnt)&(PC∉RC) -> X(newState=sKill)` (b)

`LTLSPEC G(oldState=sRomIn)&(PC∉(RC∪MC)) -> X(newState=sKill)` (c)

The LTL specification (a) checks that when the system starts/resets, it has to jump to state sRomEnt. Then, (b) checks that after the state sRomEnt, the new state is always sRomIn. Finally, (c) checks that if the current state is sRomIn and the executing PC is not within the security monitor code regions, it kills the system.

Monitor execution verification: The monitor, the other part of the security monitor, can communicate with the outside world using the defined entry points. The monitor and the root can communicate with each other freely. Otherwise, The monitor starts and finishes execution from an entry point (sMonitorEnt). As shown in figure 6-2, the state sMonitorIn is only reachable from itself (sMonitorIn), from sRomIn, and from sMonitorEnt.

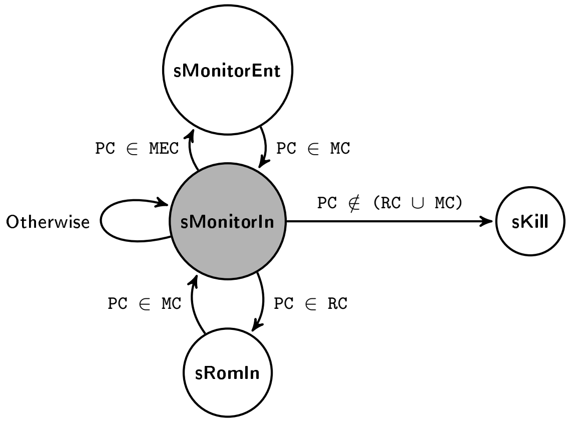


Figure 6-2: State sequence correctness starting from the state sMonitorIn

`LTLSPEC G(oldState=sMonitorIn)&(PC∉(RC∪MC)) -> X(newState=sKill)` (d)

`LTLSPEC G(oldState=sMonitorIn)&(PC∉MEC) -> X(newState=sKill)` (e)

The LTL specifications (d) and (e) check for non-authorized jumps in/from the monitor. The program should pass by the sMonitorEnt state, i.e., it should use the defined entry points.

Controlled Calls

The security monitor and the D-zones have defined entry points. If a code tries to bypass these entries and jumps to any other address inside, the EAP must raise an error. The transition step checks if the entry points were called before. This security property is proven with the previous one.

Atomicity

In this work, once the security monitor or D-zones are called, the interruptions are disabled. Attackers can use interrupts to run Time-of-check-to-time-of-use (TOCTTOU) attacks. They can program a timer to occur every t units of time, and from the interrupt handler, they could write/read memory. Handling secure interrupts is out of the scope of this thesis and will be addressed in future work.

6.1.3 Asset Protection and Confidentiality Computer-Aided Verification

When the transition from a state to another is allowed, the EAP module checks if the data memory access is authorized. Here we verify this step formally.

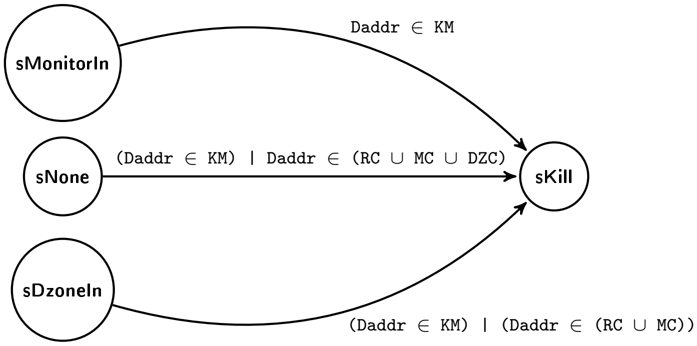

sMonitorIn
Daddr ∈ KM
sNone
(Daddr ∈ KM) | Daddr ∈ (RC ∪ MC ∪ DZC)
sKill
sDzoneIn
(Daddr ∈ KM) | (Daddr ∈ (RC ∪ MC))

Figure 6-3: Code and keys confidentiality

Key Confidentiality

If an attacker can read the keys, they can update the security monitor, for example, with a malicious one. The keys must be accessible only from the root. The EAP must raise an access error whenever a code from outside the root tries to access any key. Figure 6-3 shows that whether the system is running monitor code, D-zones code, or any other non protected code (sNone), they cannot access the cryptographic key. This results in resetting the device.

`LTLSPEC G ((newState!=sRomIn)&(Daddr∈MK) -> X reset=true)` (f)

The LTL specification (f) guarantees that whenever a program running from outside the root and tries to access the cryptographic keys, it implies a system reset. Although, an attacker can use an IO peripheral to read the corresponding memory and retrieve keys. In this case, the MMP guarantees that no IO peripheral can access the keys. Only the CPU running the root(while executing it) can access root sensitive data.

Code Confidentiality

The code is protected from unauthorized read/write. It means, as shown in Figure 6-3, critical code is kept confidential. Moreover, the MMP can guarantee that even IO peripherals can access these code regions. The LTL specifications (g1), (g2), and (g3) check for leakage attempts.

`LTLSPEC G((newState!=sRomIn)&(Daddr∈RC) -> X reset=true)`(g1)

`LTLSPEC G((newState!=sRomIn & newState!=sMonitorIn)&(Daddr∈MC)-> X reset=true)`(g2)

`LTLSPEC G ((newState=sNone)&(Daddr∈DZC)) -> X reset=true)` (g3)

Stack and Heap Confidentiality

The security monitor and the D-zones have each their own stack and heap. The EAP controls the PC before granting access to these memory regions. LTL specifications (h1) and (h2) checks if there is a reachable state where it is possible to leak from security monitor and D-zones memory regions.

`LTLSPEC G ((newState=sNone)&(Daddr∈(SMM∪DZM)) -> X reset=true)` (h1)

`LTLSPEC G ((newState=sDzone)&(Daddr∉DZM) -> X reset=true)` (h2)

Root Isolation and Immutability

root code has to be immutable. Otherwise, an attacker can change it to a malicious code and break the system security. This achievable by putting the root code within the ROM. Then the EAP ensures that the transitions to this component are respected the it was defined in Figure 6-1.

6.1.4 Software Attestation Verification

Our software attestation is built using the HACL* HMAC implementation [105]. HACL* code has been verified and proved correct, memory safe, and secret independent. On top of this, it is a part of the security monitor which is protected by the MMP and the EAP from ROP and IO based attacks. This guarantees multiple points; controlled calls of the HMAC function, code confidentiality, keys confidentiality and hash calculation intermediate values secrecy. Therefore, **SP3, SP6, SP7** and **SP8** are respected. Besides, the HMAC is precisely part of the root. It means that it is immutable, therefore it guarantees **SP2**.

Finally, concerning state sequence correctness, **SP1**, implies that the HMAC implementation conforms the defined standard specification on all possible inputs, and that it runs in a finite time. These aspects are guaranteed by HACL* implementation [105].

6.2 Toubkal Evaluation

In this section, we evaluate the silicium area (given in NAND gate equivalent) of Toubkal targeting a 90nm Low Power technology for a clock frequency of 100MHz and compare between the different options and policies. In this experiment, tests are limited to a 32 bit address space. Toubkal surface is stated in Gates Equivalents (GE). kGE refers to thousands of GE. A GE is retrieved from the surface of a NAND2 gate. We evaluate the size of each module; the MMP and the EAP, separately as Toubkal is modular and its modules can work independently.

Then, we evaluate the run-time of the root attestation and keys generation. Finally, we give the memory consumption of its code in the ROM.

6.2.1 The Master Memory Protection Footprint

Figure 6-4 gives an overview of the logical structure of MMP. The grey color represents memories, which are registers. And the white forms represent combinational bricks.

Comparison between configuration methods

Section 5.2.1 presents two different configuration methods to store Masters Id, UMLB where the slot is valid for one Master and the SMLB where the slot is valid for multiple Masters. Table 6.1 shows

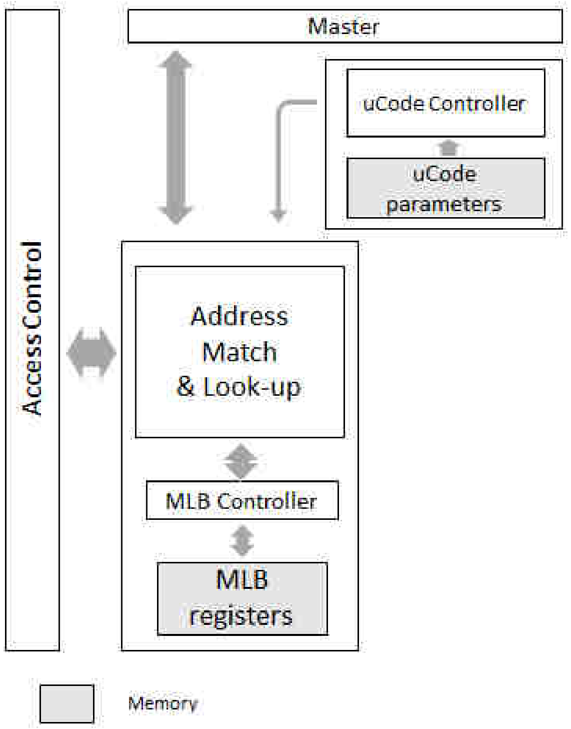


Figure 6-4: System Architecture Overview.

the impact of each method on MMP area. In the UMLB, for n Masters, MMP needs *log2Ceil(n)* bits to store all possible values of Masters Ids. While in the SMLB, MMP needs n bits because it flips the p bit to activate a region for Master number p (see section 5.2.1). However, n is greater than or equal to *log2Ceil(n)* for n strictly positive, therefore MMP needs more registers for SMLB comparing to UMLB. Here, the test was done for 2 Masters, and *log2Ceil(2)* equals 1.

Registers are not the only thing impacted. The combinational logic is impacted too. In the SMLB, MMP needs more logic than in the UMLB. To flip the p bit is more complex than just writing the value p in a register. The same is correct for the look-up&match. MMP has to look if the bit number p is up or not.

The choice between the two possibilities is strategic. In case there are more common regions between different Masters, it is more interesting to use the SMLB rather than the UMLB. This can save many slots. The use case presented in section 2.3.2 would work perfectly with the SMLB. Three Masters CPU, DMA and CE share the same region. Here the SMLB saves two slots. But in

Table 6.1: MMP area for a dynamic policy (in kGE)

	4 slots	8 slots	12 slots	16 slots
UMLB	3,36	6,21	9,10	12,04
SMLB	3,35	6,31	9,19	12,11

Table 6.2: MMP area for a dynamic policy (in kGE)

	4 slots	8 slots	12 slots	16 slots
Coarse granularity	3,36	6,21	9,10	12,04
Fine granularity	4,84	8,90	13,49	17,76

case Masters do not share regions, the UMLB is more suitable.

Comparison between coarse grained and fine grained granularities

Table 6.2 shows results of different syntheses for coarse and fine granularities. The fine grained option requires more area than the coarse grained one. The gap between the two is proportional to the number of available slots. This is due to two factors that were discussed in section 5.2.1. The first factor is the growing size of the MLB registers. For the coarse granularity, the size of a region is a power of two and is on 5 bits. While for the fine grained granularity, instead of the size of a region we have the limit of the region which is on 27 bits for a 32 bit address space and an alignment to 32 bytes. Therefore for each slot, there are 22 additional bits in the fine grained granularity. Added to this the logic behind the look-up&match. The implementation of the parallel look-up in this case is more complex and requires more logical elements to correctly answer to the timing constraints.

Nevertheless, the coarse grained option is constrained. The user needs to configure regions with a size power of two, and the base address is aligned with the size, while the fine grained requires only that the size is multiple of 32 bytes and the base address is aligned to 32 bytes.

At this level of isolation, we believe that the coarse grained option would not be very constraining for developers. Depending on the device resources, some designers would opt for the coarse grained option to save some hardware area, other would opt for the fine grained one for more user-friendliness and less engineering. However, such advantage comes with a surface cost.

Table 6.3: MMP area for different policies (in kGE)

	4 slots	8 slots	12 slots	16 slots
Static	0,15	0,16	0,18	0,21
Semi-Static	3,45	6,45	9,49	12,31
Dynamic	3,35	6,31	9,19	12,11

Table 6.4: MMP area for semi-static policy (in kGE)

Hard-coded slots	4 slots	8 slots
0 slots	3,45	6,45
1 slots	2,74	5,85
2 slots	2,01	5,16

Comparison between policies

Here, we compare between the three policies, static, semi-static and dynamic. We fix granularity into the coarse one, configuration method into the second method, then for each policy we vary the number of slots.

Table 6.3 shows the results of our syntheses. Static policy results catch our eyes immediately because of the huge gap between this policy and the others. In fact, static policy results are normal. As seen in section 5.2.1, the MLB configuration is hard-coded. Therefore, when Chisel3 generates the optimized and synthesizable Verilog code, it transforms all registers into simple wires and assigns values. And there is less and less combinational logic in MMP. Chisel3 compiler also removes all redundant values. For example if some regions have the same size value, it will just assign the value once, and use it for all the concerned regions while looking-up&matching.

Meanwhile, the gap is smaller between the semi-static and dynamic policies. The semi-static policy cells area is a little bigger than the dynamic one because MMP needs more combinational logic. In semi-static policy, slots cannot be fully modified. Developers can only modify the Masters Ids registers, and activate or deactivate the configuration for a specific Master.

For the semi-static policy, some slots can be hard-coded. For each hard-coded slot, the reduction of MMP cells area is not negligible at all. Table 6.4 shows some results. The size of MMP is reduced proportionally to the number of hard-coded slots. This reminds us of the static policy. The hard-coded slots are not stored in registers anymore. Hence, the surface area is reduced.

6.2.2 The Execution Aware Protection Footprint

The EAP does not have multiple options like the MMP. The EAP is mainly composed of high speed registers and combinational logic so it can check every instruction within the cycle. The hardware footprint of the EAP is 2,34 kGE.

The EAP size is justified by the use of high speed registers to store levels configurations. The size can be drastically reduced in case these levels configuration are hard-coded as we use less storage. However, hard-coding configurations makes the EAP less flexible.

6.2.3 The root Footprint and performance

Attestation performance: According to RISCV-VP (Virtual Prototype) simulator [41], system boot process simulation takes around 24ms to attest 8kB of data, and around 2.5ms to generate two cryptographic keys. The cryptographic keys are generated only once then stored in non-volatile memory accessible only by the root. As for the hash calculation, it is also stored with the keys. However, it is computed every system reset. The stored hash is used to check if the monitor has been updated to generate or not new cryptographic keys.

root Footprint: Toubkal requires around 4KB in the ROM. Most of the code is for storing Hacl* library. We tried to keep the ROM as small as possible to avoid software bugs. The root is stored in the ROM, so, it is immutable. Therefore, if there is any bug in the software, it cannot be patched (example of the Nintendo Switch Firmware [3, 45]). For this reason, in Toubkal, we split the monitor into two parts, the first one which is trusted and immutable is kept small, and its primary function is to check the integrity of the monitor.

6.3 Summary

In this chapter, we analysed the security provided by Toubkal and evaluated its cost. The security analysis consisted of defining security properties that Toubkal has to respect to offer the desired protection level. We represented only the EAP into an FSM; the MMP was not verified because it is a simple design and consists only of checking access right of a signal, contrary to the EAP which has a more sophisticated design with multiple states and transitions. However, the MMP was used in the security analysis to prove some security properties. Then, we used NuSMV, the symbolic model checker, to prove the correctness of these security properties.

We concluded this chapter with an evaluation of the footprint of Toubkal. We evaluated each block aside as Toubkal is modular, and each block can work independently from the others. However, as it has been said before, reducing a block reduces architecture security and can make it vulnerable to a specific type of attack. The results of the evaluation show that for a standard configuration, Toubkal requires around 5,62 kGE.

The attestation software requires around 4KB of ROM, and the attestation run-time is acceptable for lightweight devices.

Chapter 7

Toubkal design discussion and perspectives for future works

This chapter discusses Toubkal results and compares the architecture to the studied ones in chapters 3 and 4. Then, it presents future work to improve Toubkal from both performance and protection perspectives.

7.1 Discussion

In the last two chapters, 5 and 6, we presented Toubkal design, the security analysis, and the evaluation of both area and performance. In this section, we are going to discuss the obtained results.

Toubkal's goals were to achieve *flexible* isolation and attestation for lightweight devices by offering *multi-layer isolation*, a high protection level for the *security monitor* while preserving *confidentiality* secrets, and critical applications. All these features with a *low area and run-time overhead.*

Concerning the first point, which is multi-layers isolation, Toubkal introduces the MMP and the EAP to create layers of isolation and protection. The MMP can be seen as the first layer of isolation, targeting the IO peripherals. Then, the EAP can be seen as the second layer of isolation, dividing the software into four separated levels, and controls accesses and transitions. The third layer of isolation was highlighted but omitted in this work. This layer concerns the separation of the multiple applications and libraries running on the same device. The idea highlighted is to use the PMP to establish this layer. However, our study in chapter 4 shows that the PMP can introduce a colossal run-time overhead for context-switches which can be unacceptable for deterministic applications, especially for interrupt handling.

Speaking of interrupts, Toubkal does not handle interrupts in level zero and level one. This is a limitation for this work and in the next section, we will show how to address this issue securely while offering good performances comparing to the results in chapter 4.

Concerning the security monitor, it is composed of two components: an immutable root and a monitor. This work only addressed the immutable root, which is responsible for ensuring the integrity and authenticity of the rest of the software. The root offers attestation of pieces of codes and keys management. The root can attest any piece of code using a symmetric key and the HMAC from Hacl* library. The root supports three keys; K_{SM}, K_{DZ}, and K_{RC}. The first one is hard-coded, while the others are generated from the computed hash of the monitor and K_{SM}. Each key is used for encryptions and attestations in its respective level. For example, K_{SM} is only used to attest the monitor. The root uses automatically K_{DZ} to attest a piece of code in D-Zones.

The monitor component was not developed in this work and can be addressed for future work, or we can also use existing work such as MultiZone [14].

Concerning code confidentiality, as side-channel techniques are out of the scope of this work, Toubkal guarantees code confidentiality using software or IO peripherals with both the MMP and the EAP. However, this is not sufficient. Encryption of critical code would offer full confidentiality. We will discuss in the next section how to offer code encryption with hidden run-time overhead.

Then, Toubkal was validated using aided-computer formal verification. In chapter 6, we offered a detailed security analysis and validated the design. We defined security properties that Toubkal has to respect and proved them using NuSMV [21]. This helps to guarantee the correctness of the implementation of Toubkal protection features.

Finally, the evaluation shows that the hardware footprint is acceptable for lightweight devices. As for the root, most of the code footprint corresponds to the cryptographic library Hacl*, which is a verified implementation. The simulation shows an acceptable result of the attestation run-time and key generation. What would have been interesting to evaluate is the impact of context-switches and interrupts handling. However, this part was omitted from this work, as we did not implement the third layer of isolation.

7.1.1 Comparison to Existing Architectures

Here, we present a brief comparison to existing architectures. The comparison uses the same criteria defined in chapter 3. Table 7.1 summarizes the comparison.

Toubkal offers isolation too as the other studied architecture (except for VRASED and SMART). However, Toubkal offers isolation and the ability to create multiple zones even for IO peripherals, while all the others offer, at the top, limited control of IO peripherals. For example, TrustZone-M offers only a binary separation, and Sanctum offers only the possibility to configure a continuous memory area for the DMA.

Concerning attestation, Toubkal, like many, offers the possibility to attest a memory region [a, b]. The main difference is the use of a verified cryptographic library which gives more guarantees on the correctness of the security properties a system must achieve during cryptographic operations.

Toubkal offers a hybrid TCB like all the other architectures, except for Sancus. Toubkal's software TCB is composed of two components, unlike uVisor, SMART, and the likes. A first part that is immutable and responsible only for attestation, the other is partially trusted and can be updated. The second part is attested at system start and before critical operations to make sure an attacker did not tamper with.

SGX is the only architecture offering full code confidentiality. However, Toubkal, with its enforced protection of the security monitor and D-Zones, offers better partial code confidentiality as even IO peripherals' memory accesses are controlled and can be prevented from reading secret code.

Finally, for communication, this work did not offer a full IPC solution. However, the root can be called to attest that the message is coming from a trusted source. Here we have a small limitation in Toubkal, only devices with the same version of the monitor can communicate with each other because of the function generating symmetric keys. The function uses the hash of the monitor.

Table 7.1: Summary of Toubkal's comparison to the studied isolation and attestation architectures.

	Isolation	Inter-Process Communication	Root of Trust	Dynamic loading	Exception Handling	Application reboot	Attestation	Code confidentiality	IO Peripherals protection	Lightweight	Memory Protection Module	Hardware-Only TCB	Software adaptation	Open Source	Academic	Deployed	ISA
Mondrix [104]	x	x	o	o	x	o	o	o	o	o	Mondrian	o	x	o	x	o	Multiple
SMART [30]	o	o	x	o	o	o	x	o	o	x	-	o	x	o	x	o	AVR/MSP430
Sancus [67]	x	x	x	x	o	?	x	o	o	x	-	x	x	x	x	o	MSP430
SGX [61]	x	x	x	x	x	x	x	x	xo	o	MMU	o	x	o	o	x	x86_64
TyTan [18]	x	x	x	x	x	o	x	o	o	x	EA-MPU	o	x	o	x	o	Siskiyou Peak
TrustLite [48]	x	x	o	o	x	o	x	o	o	x	EA-MPU	o	x	o	x	o	Siskiyou Peak
uVisor [9]	x	x	o	o	x	o	o	o	o	x	MPU	o	x	x	o	x	Arm
TockOS [53]	x	o	o	x	x	xo	o	o	xo	x	MPU	o	x	x	x	x	Arm
Sanctum [27]	x	x	x	x	x	x	x	o	xo	o	MMU	o	x	x	x	o	RISC-V
TrustZone-M [101]	x	o	x	x	o	o	x	o	x	x	-	o	x	xo	o	x	Arm
Sopris [43]	x	o	x	o	o	?	x	o	o	x	MMU	o	x	o	o	o	Arm
EPOXY [23]	x	o	o	o	o	o	o	o	o	x	MPU	o	o	x	x	o	Arm
ACES [22]	x	o	o	o	o	o	o	o	o	x	MPU	o	o	x	x	o	Arm
MultiZone [14]	x	x	x	o	?	x	xo	o	xo	x	PMP	o	o	o	o	x	RISC-V
VRASED [68]	o	o	x	o	o	o	x	o	xo	x	-	o	x	o	x	o	AVR/MSP430
Toubkal	x	xo	x	o	o	o	x	xo	x	x	MMP&EAP	o	x	xo	xo	o	RISC-V

x: Yes, o: No, xo: Partial, ?: NA, -: Non-relevant

7.2 Future Work

This section will investigate design perspectives to improve protection and performance. We present approaches that were studied but not implemented in this work or alternatives that can be used in Toubkal to accomplish certain protection guarantees.

Mainly, we will discuss how upgrading the EAP and merging it with the PMP will offer more advanced access control with better performance. How we can establish remote attestation to attest devices and make sure they are not compromised before exchanging data. And how to provide full confidentiality for critical applications with a hidden run-time cost.

7.2.1 Enforced Access Control

To enforce access control of the running application and improve performance comparing to results in chapter 4, we propose to work around the context-switch mechanism. The current PMPs or MPUs require a software context switch. However, depending on the complexity of applications and contexts configurations, this step can take much time to finish. Therefore, it can impact on the deterministic characteristic of lightweight devices.

Before beginning the discussion on improving the design of the PMP and the EAP in order to improve context-switches and interrupts handling, we explain how it works in most cases. context-switches happen when the system changes the execution zone, and this can either happen with a simple call to the gateway or with an interrupt rise. Figure 7-1 illustrates the steps to switch the context.

The context-switch happens in four steps. The first step is the gateway in. The second step is the execution of the called function in the new context. The third step is the gateway out. Finally, the fourth step is the return to the caller function.

The gateway in main function is to save the running context; stack pointer and PMP configuration, load the new context; stack pointer and reconfigure the PMP, and clean if there is anything sensitive. The operation starts from a non-privileged code, escalates the privilege with a software interrupt, then de-privileges to start the second step.

The gateway out consists of restoring the context of the caller; stack pointer and reconfiguring the PMP, then de-privileging the execution to return back to the caller execution.

So, the operations overwhelming the context-switche are the gateway in and the gateway out.

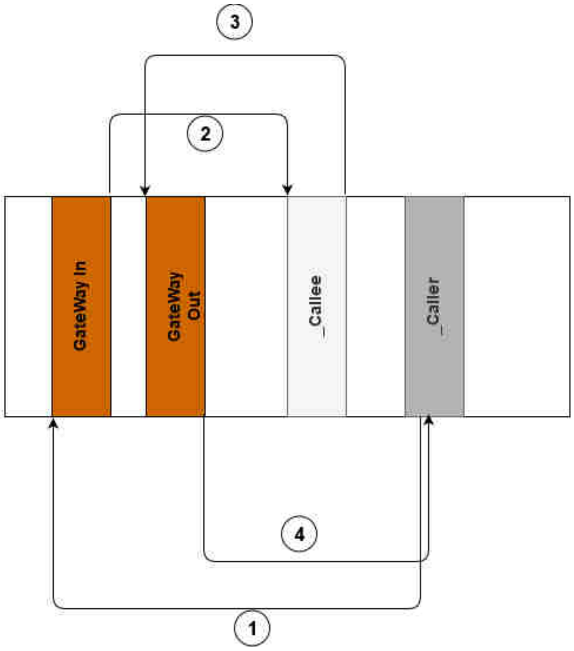


Figure 7-1: context-switche main steps: 1) The _Caller calls the gateway of the _Callee. The gateway in escalates the privilege, saves the _Caller context and loads the _Callee context. 2) The gateway de-privileges the execution and jumps to the _Callee entry. The _Callee does its job. Then, it calls the gateway out. 3) The gateway out escalates the privilege again, loads the stored context of _Callee. 4) The gateway out de-privileges the execution and jumps to the return address in the _Caller.

The system has to reconfigure all the PMP slots among other configurations and safety checks. Our idea is to make the context-switche smoother than this way by making the PMP and EAP more context-aware, and move a part of the context-switche into the hardware.

We propose to introduce for each context a hardware unique identifier HID. This HID is used to know which PMP configuration to apply. So instead of a software configuring the PMP each context-switche, we can have a small Content-Addressable Memory-like (CAM) within the PMP and look for the PMP configuration of the running HID. First, by doing so, we do not need anymore to reconfigure the PMP each context-switche. Second, if the configuration is not found within the PMP, the PMP will use the CAM-like to see if there is any. The software will only be responsible for changing the HID instead of updating all the PMP slots. This way is faster than the software-only alternative.

Another possibility is to create entry points linked to each HID. However, this may increase the hardware drastically, but may improve drastically context-switches performance and protection.

The HID can also be useful to create more access control. The EAP can use it to create a white-list of contexts that are authorized to call the security monitor or D-zones services. This way, the EAP controls better the transitions to critical applications.

7.2.2 Enforced Attestation

Lightweight devices are highly connected. They exchange data with multiple devices. To ensure trust between devices before starting communicating, we use remote attestation. Here, we describe how two devices attest, mutually, that they are not fully compromised.

Suppose we have a device *A* and a device *B*. *A* wants to send data to *B*. *A* and *B* care only about the integrity of the data and not the confidentiality. Therefore, only attacks to change the data shared in order to change the behaviour of device B or inject malicious software are considered.

Here, $_{\mathtt{A}}\mathtt{Send}_{\mathtt{B}}\mathtt{(D)}$ means device *A* sends data to device *B*. `Hmac(X,Y)` means we compute an hmac of X with the key Y. Therefore, Y has to be 256 bit, and the computation result is 256 bit length.

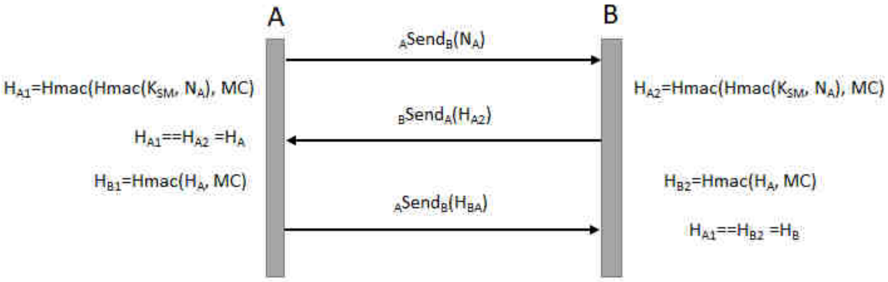

A
B
$_{A}Send_{B}(N_{A})$
H_{A1}=Hmac(Hmac(K_{SM}, N_{A}), MC)
H_{A2}=Hmac(Hmac(K_{SM}, N_{A}), MC)
$_{B}Send_{A}(H_{A2})$
H_{A1}==H_{A2} =H_{A}
H_{B1}=Hmac(H_{A}, MC)
H_{B2}=Hmac(H_{A}, MC)
$_{A}Send_{B}(H_{BA})$
H_{A1}==H_{B2} =H_{B}

Figure 7-2: Remote attestation scheme.

Figure 7-2 shows the different steps to establish mutual trust between devices *A* and *B*. First, *A* initiates the communication by sending a nonce N_A to *B*. Then, both *A* and *B* compute $\mathtt{H_{AA}}$ = $\mathtt{Hmac(Hmac(}K_{SM}\mathtt{,N_A),\ MC_A)}$ and $\mathtt{H_{AB}}$ = $\mathtt{Hmac(Hmac(}K_{SM}\mathtt{,N_A),\ MC_B)}$ respectively, where MC is each one's monitor code. Normally, it is the same, except if it was compromised. At this stage, we do not know.

Then, device *B* sends $\mathtt{H_{AB}}$ to *A*. *A* compares it to $\mathtt{H_{AA}}$. If the check passes, *A* can trust *B*. However, *B* cannot trust *A* yet. Then again, both devices *A* and *B* compute $\mathtt{H_{BA}}$ = $\mathtt{Hmac(H_{AA},\ MC_A)}$ and $\mathtt{H_{BB}}$ = $\mathtt{Hmac(H_{AB},\ MC_B)}$ respectively. Device *A* sends the result $\mathtt{H_{BA}}$ to *B*, and *B* compares it to $\mathtt{H_{BB}}$. If they match, then *B* can trust *A*.

According to the results in chapter 6, this operation is still acceptable in most cases. However, it really depends on the size of the monitor. Here again, devices need to embed the same version of the monitor.

7.2.3 Code Confidentiality

Here, we explain how we can support full code confidentiality in Toubkal. We propose loading into the Flash encrypted critical codes and hide the cost of decryption during run-time by extending the MMP. The MMP catches signals from all IO peripherals, and this includes the CPU itself.

We need to add some storage to the MMP module and some logic. We also need a symmetric cryptographic hardware module. Finally, we need to allocate a secure memory region in the RAM. This memory region can be statistically or dynamically allocated. However, if it is allocated dynamically, the developer has to make sure that no sensitive data may be leaked.

We need another bit in the MLB, so when we configure a memory region in the MMP, we chose if the region is encrypted or not. We also need storage for the encryption key. There is the possibility of supporting multiple keys. Then, we can add in the MLB a few bits to choose the key ID that will be used to decrypt the code.

When the CPU is executing an instruction for the first time, the instruction signal needs to go by the MMP. Therefore, we also need to connect the instruction bus to the MMP. The MMP will catch the signal. Then, the MMP will check for access right, and also it will check if the memory region is encrypted or not. In the case it is encrypted, the MMP will send a signal to the cryptographic block which is connected with the Flash. This module will retrieve encrypted data, depends if it is 128 bit or more, decrypts it, then stores the result in the allocated memory in the RAM. This way, Toubkal will hide the decryption cost for future instruction. The cryptographic module will keep decrypting until the end of the critical code, or when the allocated memory is full.

Once we finish the execution of the critical code, we clean the allocated memory region for future use. The cryptographic module is the only one that can write in that memory region, and the CPU will be able to fetch data from it when it is trying to execute the critical code.

7.3 Summary

In this chapter, we discussed the design of Toubkal, compared it to the studied architectures, and presented some future work to improve protection features.

The discussion shows that Toubkal, comparing to the other architectures, offers in-depth isolation that was neglected, especially in lightweight devices. However, there are other aspects of Trusted Computing that were not addressed in this work, such as interrupts handling and full code confidentiality.

The perspectives present three improvements for Toubkal that we think are important. These improvements concern context-switches and interrupt handling, remote attestation, and full code confidentiality. We presented designs that are not implemented yet and can extend the design of Toubkal. We also gave an estimation of the area or the run-time overhead of each feature.

Chapter 8

Conclusion

The increased use of highly connected lightweight devices leads to a significant scale of attacks. These devices are part of our daily life. Recent attacks prove the need to create secure devices out of fear that they leak our private information or having physical damage to their surroundings. This thesis proposed an in-depth study of isolation and attestation architectures, then, proposed a security architecture to help to reduce the risk of attacks.

First, we studied existing isolation and attestation architectures to understand what works and what does not. Why some are deployed, but the majority are not. The study showed a variety of interesting architectures offering different security features targeting a variety of applications. Concerning high-computing architectures, there are some architectures with a good protection level, and they are already deployed. However, the case of lightweight devices is different. Lightweight devices are constrained devices and time-critical. Therefore, it is more challenging to establish an excellent protection level under these circumstances. Architectures targeting lightweight devices failed at least in one of the following criteria: performance, security protection, cost, and flexibility.

Toubkal's primary goal is to provide a low-cost modular and highly secure architecture with good performance. To improve the security protection of existing architecture, Toubkal provides multi-layer isolation with attestation. In this work, we presented two of the three layers needed to establish in-depth isolation. The first layer is the isolation of IO peripherals. The second one concerns the creation of the separated level to enforce separation even in the privileged mode. Finally, the third level, that was not handled here, is the one established in all existing architectures.

To establish the first layer of isolation, Toubkal offers the MMP, which is responsible for controlling signals passing through the interconnection bus. The need for this layer of isolation is justified by the recent attacks targeting these peripherals to access memory regions that are not, at first glance, accessible in the running state of the device. Existing isolation architectures target only memory protection at the CPU level, and IO peripherals bypass CPU protection.

To establish the second layer of isolation, Toubkal provides the EAP, which is a hardware module connected to the core. The EAP divides the software into four main levels and controls memory regions for each level and transitions from one level to another. The primary goal of this module is to create a real separation of the security monitor from all the rest. The need for this layer of isolation is justified by the fact that OSes run in high privileged mode, same as the security monitor, and they add a large attack surface. This makes the system very vulnerable to privilege escalation attacks.

However, isolation alone is not sufficient for a security architecture in a highly connected device. If an attacker succeeds in breaking inside an isolated level or application, they still can control that level or application. Therefore, Toubkal offers the possibility to check the state of each level or application to make sure it was not compromised. Toubkal provides an immutable root using an HMAC algorithm to attest every piece of code or data. This way, the system can verify the integrity and authenticity of each level and application.

Besides the security features, Toubkal was validated with aided-computer formal verification methods. We defined security properties concerning the isolation execution and protection and confidentiality of assets. Then, we proved these properties using formal verification methods. We also evaluated the overhead introduced by Toubkal. The evaluation showed that the overhead is acceptable for lightweight devices.

Finally, we concluded with an analysis of where Toubkal stands, of what is missing in this work and how we can improve it. We have shown that there has been much interesting work and that there is still much room for improvement.

Bibliography

[1] Cadence software. https://www.cadence.com/.

[2] Freertos tcp/ip stack vulnerabilities put a wide range of devices at risk of compromise: From smart homes to critical infrastructure systems, 2018.

[3] Shofel2 exploit. https://github.com/fail0verflow/shofel2, 2018.

[4] Intel. 2017. Intel control-flow enforcement technology preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017.

[5] Anati, Shay Gueron, Simon Paul Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing ittai. 2013.

[6] ARM. Cortex-m3 devices generic user guide, 2010.

[7] ARM. Arm amba 5 ahb protocol specification, 2015.

[8] ARM. Arm the architecture for the digital world. https://www.arm.com/, 2015.

[9] ARM. uvisor. https://github.com/armmbed/uvisor, 2015.

[10] ARM. Arm system memory management unit architecture specification, 2016.

[11] ARM. Trusted firmware-m. https://git.trustedfirmware.org/trusted-firmware-m.git/refs/, 2017.

[12] Tech ARM. Arm®v7-m architecturereference manual, 2006.

[13] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing hardware in a scala embedded language. *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

[14] Donald Barnetson. How to secure a risc-v embedded system in just 30 minutes. 2019.

[15] Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis, 2008.

[16] Muli Ben-Yehuda and Karl Rister. The price of safety : Evaluating iommu performance. 2007.

[17] El Mehdi Benhani, Cédric Marchand, Alain Aubert, and Lilian Bossuet. On the security evaluation of the arm trustzone extension in a heterogeneous soc. *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 108–113, 2017.

[18] Franz Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: Tiny trust anchor for tiny devices. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[19] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. *2010 IEEE Second International Conference on Social Computing*, pages 768–775, 2010.

[20] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault attacks on encrypted general purpose compute platforms. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 197–204, New York, NY, USA, 2017. ACM.

[21] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, 2002.

[22] Abraham A. Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *USENIX Security Symposium*, 2018.

[23] Abraham A. Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303, 2017.

[24] ATMEL CORPORATION. Atmel trusted platform module at97sc3201, 2005.

[25] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *USENIX Security Symposium*, 2018.

[26] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[27] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[28] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: Hardware-assisted flow integrity extension. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[29] Ruan de Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (cfi). *CoRR*, abs/1706.07257, 2017.

[30] Karim M. El Defrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, 2012.

[31] Easycrypt. Easycrypt: Computer-aided cryptographic proofs. https://www.easycrypt.info/trac/, 2009.

[32] eChronos. echronos. https://ts.data61.csiro.au/projects/TS/echronos/, 2018.

[33] A. Kegel et al. Input/output memory management unit with protection mode for preventing memory access by i/o devices. (US8631212), 2014.

[34] M. Hummel et al. Direct memory access (dma) address translation in an input/output memory management unit (iommu). (US7809923B2), 2010.

[35] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry S. Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Flexible hardware-managed isolated execution: Architecture, software support and applications. *IEEE Transactions on Dependable and Secure Computing*, 15:437–451, 2018.

[36] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry V. Ponomarev, Nael B. Abu-Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202, 2014.

[37] Aurélien Francillon, Quan Hoang Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. *2014 Design Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2014.

[38] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix C. Freiling, and Ingrid Verbauwhede. Soteria: Offline software protection within low-cost embedded devices. In *ACSAC*, 2015.

[39] James Greene. Intel ® trusted execution technology hardware-based technology for enhancing server platform security. 2013.

[40] GreenIPCore. Amba-ahb security, 2018.

[41] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. Extensible and configurable RISC-V based virtual prototype. pages 5–16, 2018.

[42] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ISCA*, 2013.

[43] Galen Hunt, George Letey, and Ed Nightingale. The seven properties of highly secure devices, March 2017.

[44] Paul A. Karger and Roger R. Schell. Thirty years later: Lessons from the multics security evaluation. In *ACSAC*, 2002.

[45] Mikaela Szekely Kate Temkin. Fusee gelee exploit. https://nvd.nist.gov/vuln/detail/CVE-2018-6242, 2018.

[46] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, April 1999.

[47] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.

[48] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *EuroSys*, 2014.

[49] Ram Kumar, Eddie Kohler, and Mani B. Srivastava. Harbor: Software-based memory protection for sensor nodes. *2007 6th International Symposium on Information Processing in Sensor Networks*, pages 340–349, 2007.

[50] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.

[51] Steve Langan. Technical report, 2017.

[52] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. *2014 IEEE Symposium on Security and Privacy*, pages 276–291, 2014.

[53] Amit A. Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *SOSP*, 2017.

[54] Amit A. Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In *APSys*, 2017.

[55] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[56] LLVM. Add support for embedded position-independent code (ropi/rwpi). https://lists.llvm.org/pipermail/llvm-dev/2015-December/093022.html, 2015.

[57] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Thomas Müller, Felix C. Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67:361–374, 2018.

[58] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.

[59] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. *2010 IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[60] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In *EuroSys*, 2008.

[61] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[62] Kenneth L. McMillan. Symbolic model checking. 1993.

[63] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

[64] Daniel Münch, Michael Paulitsch, Oliver Hanka, and Andreas Herkersdorf. Mpiov: Scaling hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (multi-core) multi-processor systems. *2015 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 579–584, 2015.

[65] Daniel Münch, Michael Paulitsch, and Andreas Herkersdorf. Iompu: Spatial separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non-transparent bridges. *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 1037–1044, 2015.

[66] Fredrik Nilsson and Niklas Adolfsson. A rust-based runtime for the internet of things, 2017.

[67] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, 2013.

[68] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, Santa Clara, CA, August 2019. USENIX Association.

[69] NVD. Nvd. https://nvd.nist.gov/, 2015.

[70] Dorottya Papp, Zhendong Ma, and Levente Buttyán. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152, 2015.

[71] Steffen SCHULZ Patrick Koeberl. Execution-aware memory protection, U.S. Patent US20150032996A1, Jul. 2013.

[72] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51:130:1–130:36, 2019.

[73] Inc. Qualcomm Technologies. Pointer authentication on armv8.3., 2017.

[74] Hans Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Philip England, Chris Fenner, Kinshuman Kinshumann, Jork Löser, Dennis Mattoon, Marcus Nystrom, Dana Robinson, Rob Spiger, and Stefan Thom. ftpm : A firmware-based tpm 2 . 0 implementation. 2015.

[75] Fernand Lone Sang, Vincent Nicomette, and Yves Deswarte. I/o attacks in intel pc-based architectures and countermeasures. *2011 First SysSec Workshop*, pages 19–26, 2011.

[76] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, and David Hely. Shcot: Secure (and verified) hybrid chain of trust to protect from malicious software in lightweight devices. In *The 1st Annual International Workshop on Software Hardware Interaction Faults, co-located with ISSRE 2019*, Berlin, Germany, October 2019.

[77] Abderrahmane Sensaoui, Oum-El-Kheir Aktouf, David Hely, and Stephane Di Vito. An in-depth study of mpu-based isolation techniques. *Journal of Hardware and Systems Security*, Nov 2019.

[78] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Hardware-based isolation and attestation architecture for a risc-v core. In *SiFive's Technical Symposium*, Grenoble, France, May 2019.

[79] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Poster: Hardware-based isolation and attestation architecture for a risc-v core. In *2019 CySep and EuroS&P*, Stockholm, Sweden, June 2019.

[80] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Poster: Hardware-based isolation and attestation architecture for a risc-v core. In *2019 15th European Dependable Computing Conference (EDCC)*, Naples, Italy, September 2019.

[81] Abderrahmane Sensaoui, David Hely, and Oum-El-Kheir Aktouf. Toubkal: A flexible and efficient hardware isolation module for secure lightweight devices. In *2019 15th European Dependable Computing Conference (EDCC)*, Naples, Italy, September 2019.

[82] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP*, 2005.

[83] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Swatt: software-based attestation for embedded devices. *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282, 2004.

[84] Rui Shu, Peipei Wang, Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Comput. Surv.*, 49:50:1–50:37, 2016.

[85] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Comput. Surv.*, 49(3):50:1–50:37, October 2016.

[86] SiFive. Sifive e31 core complex manual, 2017.

[87] Henry Cook SiFive. Diplomatic design patterns : A tilelink case study. 2017.

[88] Inc. SiFive. The risc-v instruction set manual, 2017.

[89] Inc. SiFive. Sifive tilelink specification, 2017.

[90] et.al Simon Johnson. Intel® sgx: Intel® epid provisioning and attestation services, 2016.

[91] Yang Song. On control flow hijacks of unsafe rust. 2017.

[92] Eugene H. Spafford. The internet worm incident. In *ESEC*, 1989.

[93] Eugene H. Spafford. The internet worm program: an analysis. In *CCRV*, 1989.

[94] Oliver Stecklina, Peter Langendörfer, and Hannes Menzel. Design of a tailor-made memory protection unit for low power microcontrollers. *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 225–231, 2013.

[95] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *DIMVA*, 2012.

[96] Raoul Strackx and Frank Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In *ACM Conference on Computer and Communications Security*, 2012.

[97] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In Sushil Jajodia and Jianying Zhou, editors, *Security and Privacy in Communication Networks*, pages 344–361, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[98] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[99] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Xiaodong Song. Sok: Eternal war in memory. *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[100] Arm Tech. Psecurity technology building a secure system using trustzonetechnology, 2009.

[101] ARM Tech. Memory protection unit (mpu), 2016.

[102] Tock. Tockos. https://github.com/helena-project/tock, 2015.

[103] Emmett Witchel. Mondriaan memory protection. 2004.

[104] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: memory isolation for linux using mondriaan memory protection. In *SOSP*, 2005.

[105] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *ACM Conference on Computer and Communications Security*, 2017.