Design of a secure kernel for constrained devices

Nicolas Dejon

Université de Lille

orange™

Université de Lille
Orange

École doctorale **Ecole Graduée MADIS-631**

Laboratoire **CRIStAL**

Thèse par **Nicolas Dejon**

Soutenue le **14 décembre 2022**

Mémoire présenté en vue de l'obtention du grade de

Docteur de l'Université de Lille

Discipline **Informatique**

Spécialité **Informatique et applications**

Conception d'un noyau sécurisé pour objets contraints

Thèse dirigée par	Gilles Grimaud	Directeur de thèse
	Julien Cartigny	Co-directeur de thèse
	Chrystel Gaber	Co-encadrante

Composition du jury

Rapporteurs	Didier Donsez	Professeur des Universités à l'Université de Grenoble-Alpes, LIG	
	Jean-Pierre Talpin	Directeur de recherche à l'INRIA Rennes	
Examinateurs	Romain Rouvoy	Professeur des Universités à l'Université de Lille, CRIStAL	(Président)
	Marie-Laure Potet	Professeure des Universités à l'Université de Grenoble-Alpes, VERIMAG	
	Julien Sopena	Maître de conférences à Sorbonne Université, LIP6	
	Patricia Mouy	Chercheuse-Docteure à l'ANSSI	
Invité	Jean-Philippe Wary	Responsable de programme de recherche à Orange Innovation	
Direction de thèse	Gilles Grimaud	Professeur des Universités à l'Université de Lille, CRIStAL	
	Chrystel Gaber	Ingénieure de Recherche-Docteure à Orange Innovation Caen	

Colophon

Manuscrit de thèse intitulé « Conception d'un noyau sécurisé pour objets contraints », écrit par Nicolas Dejon, composé au moyen du système de préparation de document LaTeX et de la classe `yathesis` dédiée aux thèses préparées en France.

Université de Lille

orange™

Université de Lille
Orange

École doctorale **Ecole Graduée MADIS-631**

Laboratoire **CRIStAL**

Thèse par **Nicolas Dejon**

Soutenue le **14 décembre 2022**

Mémoire présenté en vue de l'obtention du grade de

Docteur de l'Université de Lille

Discipline **Informatique**

Spécialité **Informatique et applications**

Conception d'un noyau sécurisé pour objets contraints

Thèse dirigée par	Gilles Grimaud	Directeur de thèse
	Julien Cartigny	Co-directeur de thèse
	Chrystel Gaber	Co-encadrante

Composition du jury

Rapporteurs	Didier Donsez	Professeur des Universités à l'Université de Grenoble-Alpes, LIG	
	Jean-Pierre Talpin	Directeur de recherche à l'INRIA Rennes	
Examinateurs	Romain Rouvoy	Professeur des Universités à l'Université de Lille, CRIStAL	(Président)
	Marie-Laure Potet	Professeure des Universités à l'Université de Grenoble-Alpes, VERIMAG	
	Julien Sopena	Maître de conférences à Sorbonne Université, LIP6	
	Patricia Mouy	Chercheuse-Docteure à l'ANSSI	
Invité	Jean-Philippe Wary	Responsable de programme de recherche à Orange Innovation	
Direction de thèse	Gilles Grimaud	Professeur des Universités à l'Université de Lille, CRIStAL	
	Chrystel Gaber	Ingénieure de Recherche-Docteure à Orange Innovation Caen	

Université de Lille

orange™

Université de Lille
Orange

Doctoral School **Ecole Graduée MADIS-631**

Laboratory **CRIStAL**

Thesis by **Nicolas Dejon**

Defended on **December 14, 2022**

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy of Université de Lille

Academic Field **Computer Science**

Specialty (in French) **Informatique et applications**

Design of a secure kernel for constrained objects

Thesis supervised by	Gilles Grimaud	Supervisor
	Julien Cartigny	Co-Supervisor
	Chrystel Gaber	Co-Monitor

Committee members

Referees	Didier Donsez	Professor at Université de Grenoble-Alpes, LIG	
	Jean-Pierre Talpin	Senior Researcher at INRIA Rennes	
Examiners	Romain Rouvoy	Professor at Université de Lille, CRIStAL	(Chair)
	Marie-Laure Potet	Professor at Université de Grenoble-Alpes, VERIMAG	
	Julien Sopena	Associate Professor at Sorbonne Université, LIP6	
	Patricia Mouy	PhD, Researcher at ANSSI	
Guest	Jean-Philippe Wary	Research program director at Orange Innovation	
Supervisors	Gilles Grimaud	Professor at Université de Lille, CRIStAL	
	Chrystel Gaber	PhD, Research engineer at Orange Innovation Caen	

The research leading to these results were funded by ANRT Convention Cifre n°2020/0380 and contributed to the TinyPART project funded by the MESRI-BMBF German-French cybersecurity program under grant agreements n°ANR-20-CYAL-0005 and 16KIS1395K. This thesis reflects only the author's views. ANRT, MESRI and BMBF are not responsible for any use that may be made of the information it contains.

A ma famille

A la mémoire de ceux qui m'ont précédé

A mes rêves d'enfant

Parvenu à l'issue de mon doctorat en informatique, et ayant ainsi pratiqué, dans ma quête du savoir, l'exercice d'une recherche scientifique exigeante, en cultivant la rigueur intellectuelle, la réflexivité éthique et dans le respect des principes de l'intégrité scientifique, je m'engage, pour ce qui dépendra de moi, dans la suite de ma carrière professionnelle quel qu'en soit le secteur ou le domaine d'activité, à maintenir une conduite intègre dans mon rapport au savoir, mes méthodes et mes résultats.

Serment doctoral d'intégrité scientifique

With the completion of my doctorate in computer science, in my quest for knowledge, I have carried out demanding research, demonstrated intellectual rigour, ethical reflection, and respect for the principles of research integrity. As I pursue my professional career, whatever my chosen field, I pledge, to the greatest of my ability, to continue to maintain integrity in my relationship to knowledge, in my methods and in my results.

Scientific integrity oath

La chance ne sourit qu'aux esprits bien préparés.
Luck favors the prepared mind.

Louis Pasteur

Si tout est connecté, tout peut être piraté.
If everything is connected, everything can be hacked.

Ursula von der Leyen

Je suis amoureuse d'un OS.
I'm in love with an OS.

Film *Her* / *Her* movie

L'OS ne fait que mentir.
The OS is lying to you.

Chrystel Gaber

Il ne faut jamais sous-estimer le pouvoir du petit canard jaune qui fait oui-oui.
Never under-estimate the power of the small yellow duck nodding his head.

Gilles Grimaud

Design of a secure kernel for constrained objects

Abstract

This thesis invests the field of cybersecurity for small computer systems (embedded systems/-connected objects/low-end devices, of type microcontroller) and more precisely aims to bring strong memory isolation guarantees for tasks executing on them.

The heterogeneity and strong resource constraints (memory, computing power, energy) of constrained embedded systems require tailored solutions. The embedded software life cycle and the specific hardware platforms challenge us to reconsider the security schemes that leave open memory vulnerability issues, still prevailing today. Furthermore, the risk of vulnerability exploits elevates with the growing number of use cases (smart environments in general) implying increased complexity within these systems and with the burgeoning market of the Internet-of-Things (notably for remote update purposes). As a consequence, cyber attackers can take profit from these vulnerabilities to take remote control of these connected systems in a very scalable way.

In this context, the thesis proposes to design a kernel for constrained objects that is able to bring strong memory isolation guarantees. It studies the blend of high flexibility and strong security for this class of devices with the aim of security-by-design without functional loss. The thesis presents two main contributions dealing with software attacks on memory.

The first contribution is an Operating System (OS) kernel, named Pip-MPU, that offers a hardware-based isolation solution with a degree of flexibility outperforming current solutions. Pip-MPU is adapted from the Pip protokernel initially designed for high-end/general-purpose computers that are provided with more furnished and different hardware platforms than the ones of constrained objects. For that, the designed kernel is a complete refactoring of Pip and offers a security mechanism based on the Memory Protection Unit (MPU), a unit of the processor, which enables hardware-based access control on memory resources. Despite strong limitations due to the limited hardware platform, Pip-MPU is as flexible as its parent project Pip. With a code base size of less than 10 Kb and about 16% extra costs in terms of performance and energy consumption, Pip-MPU reduces the number of privileged operations by 99% and the attack surface of the accessible application memory by 98%.

The second contribution is the demonstration of strong isolation guarantees by the use of formal methods. Several kernel services have been proved against isolation by the use of the Coq Proof Assistant. The proved properties are Pip's security properties that enforce a strict memory isolation security model. To our knowledge, no state-of-the-art solution offering MPU-based isolation has been proved before. We develop novel proof conduct techniques and propose new metrics to follow the proof effort and evaluate the hypotheses supporting the proofs.

All the contributions developed in this thesis are publicly released under open-source licences.

Keywords: memory isolation, constrained objects, mpu, formal verification, coq, pip, security-by-design

Conception d'un noyau sécurisé pour objets contraints

Résumé

Cette thèse s'inscrit dans la thématique de la sécurité des petits systèmes informatiques (systèmes embarqués/objets connectés, de type microcontrôleur) et plus précisément vise à apporter des fortes garanties d'isolation mémoire pour les tâches qui s'y exécutent.

L'hétérogénéité et les fortes contraintes en ressources (espace mémoire, puissance de calcul, énergie) nécessitent la mise en place de solutions sur mesure pour les systèmes embarqués contraints. Le cycle de vie des logiciels embarqués et les architectures matérielles spécifiques imposent de repenser la manière de mettre en oeuvre la sécurité qui, encore aujourd'hui, laisse ouvertes des problématiques de vulnérabilité mémoire. De plus, les risques d'exploitation de ces vulnérabilités grandissent avec l'émergence de nouveaux cas d'utilisation (environnements intelligents de manière générale) impliquant des systèmes de plus en plus complexes et l'explosion du nombre de systèmes connectés (notamment pour des besoins de mise à jour à distance). En conséquence, des cyber-attaquants peuvent tirer profit de telles vulnérabilités pour prendre le contrôle à distance de ces systèmes connectés de façon massive.

Dans ce cadre, la thèse propose d'élaborer un noyau destiné aux petits objets qui soit capable d'apporter des garanties fortes d'isolation mémoire. Elle étudie l'association entre flexibilité élevée et forte sécurité au sein d'objets contraints pour une sécurité dès la conception sans perte fonctionnelle. Elle est constituée de deux contributions principales qui répondent aux attaques logicielles sur la mémoire.

La première contribution est un noyau de système d'exploitation, appelé Pip-MPU, qui propose une solution d'isolation ancrée dans le matériel et offrant une flexibilité dépassant les solutions actuelles. Pip-MPU est adapté du protonoyau Pip initialement destiné à des ordinateurs généralistes dotés d'une plateforme matérielle plus fournie et différente de celle des objets contraints. Pour cela, le noyau conçu est une refonte totale de Pip et propose un mécanisme de sécurité basé sur la *Memory Protection Unit (MPU)*, une unité du processeur, qui permet un contrôle d'accès matériel aux ressources. Malgré les fortes limitations imposées par la plateforme matérielle, Pip-MPU est aussi flexible que ce que permet la MMU en termes de sécurité. Du haut de ses 10 Ko de code et 16% de surcoût en termes de performances et de consommation d'énergie, Pip-MPU réduit le nombre d'opérations privilégiées exécutées de 99% et la surface d'attaque de la mémoire accessible depuis l'application de 98%.

La deuxième contribution est l'obtention de garanties fortes de l'isolation par l'usage de méthodes formelles. Plusieurs services du noyau ont été formellement prouvés pour l'isolation à l'aide de l'assistant de preuve Coq. Les propriétés prouvées sont les propriétés de sécurité de Pip imposant son modèle de sécurité d'isolation stricte. A notre connaissance, aucun autre système de l'état de l'art proposant de l'isolation par MPU n'a été formellement prouvé auparavant. Nous développons des nouvelles techniques de conduite de preuve et proposons de nouvelles métriques permettant de suivre l'effort de preuve et d'évaluer les hypothèses soutenant les preuves.

Toutes les contributions de la thèse sont en source ouverte.

Mots clés : isolation mémoire, objets contraints, mpu, vérification formelle, coq, pip, sécurisé dès la conception

Remerciements

Ce document présente des travaux originaux qui n'auraient pas pu voir le jour sans l'implication d'un grand nombre de personnes que je souhaite ici remercier.

Merci à Didier Donsez et Jean-Pierre Talpin, qui m'ont fait l'honneur d'accepter de rapporter ma thèse, ainsi qu'à Marie-Laure Potet, Romain Rouvoy, Julien Sopena et Patricia Mouy pour son examen, et Jean-Philippe Wary, d'avoir accepté de faire partie de mon jury.

Cette thèse n'aurait pas existé sans l'insufflation créatrice et engagée de mes encadrants de thèse Chrystel Gaber et Gilles Grimaud, merci de m'avoir donné la chance de participer à votre vision de recherche. Merci pour votre encadrement bienveillant, pour les années de conseil et d'apprentissage, pour nos discussions, parfois philosophiques, et pour la liberté, l'autonomie et la confiance que vous m'avez accordées pour réaliser mes travaux dans les meilleures dispositions possibles, malgré la distance et la logistique inhérentes à cette thèse.

Un grand merci à mes deux équipes de recherche, à Caen et à Lille, et mes managers, de m'avoir si bien accueilli même pour des séjours (tout le temps) temporaires. Plus que des collègues, j'y ai trouvé des amis. Particulièrement, merci à David et Samuel de l'équipe 2XS de CRIStAL de m'avoir ouvert au monde de la vérification formelle et d'avoir joué les 'petits canards jaunes qui font oui-oui' pour médiatiser mes pensées et me redonner confiance dans mes intuitions. Merci à Clément, Florian (chauffeur privé, conseiller des preuves formelles, et bravo encore pour le karting), Olivier (la force utcéenne), Sofía, mes co-doctorants de Lille, et plus largement toute l'équipe 2XS (merci Thomas pour le piano) pour la chaleureuse ambiance et nos belles conversations autour de (l'ancien) frigo et (l'ancien) canapé. J'aimerais tout particulièrement remercier Narjes, ancienne membre de l'équipe 2XS, pour sa formation expresse de Pip et des preuves réalisée en tout début de thèse. Les travaux que je présente ont mis en musique un grand nombre de ses apprentissages et réalisations. Un grand merci à Damien d'avoir vaillamment repris mes travaux pour en donner la substance prototypique. Un merci particulier à Samuel de l'équipe SASE d'Orange Innovation Caen, et les stagiaires de notre petite équipe Ghilas, Idir, Matthieu et Gregory, d'avoir cogité près de moi et qui m'ont permis de nourrir mon processus de réflexion.

Merci à Orange de m'avoir donné l'opportunité matérielle d'effectuer ma thèse et de m'avoir proposé de continuer un travail de recherche après ma thèse. Orange m'a aussi donné la possibilité de défendre mes travaux dans un concours de vulgarisation au grand public qui me tenait à coeur, je l'en remercie. Merci au laboratoire CRIStAL de l'Université de Lille de m'avoir accueilli et de financer mon déplacement à l'étranger pour présenter mes travaux. J'en profite pour remercier l'ensemble de mes enseignants,

tous domaines confondus, du collage de pastille en maternelle à la rédaction de ce manuscrit, en passant par la musique, le sport, les pompiers et tant d'autres activités qui m'ont forgé, bousculé, étiré toutes ces années. Cette thèse est la quintessence de leur investissement et de la transmission de leur passion, merci.

De plus, ces travaux ont été soutenus, soumis, challengés, améliorés, discutés, présentés, exprimés, développés, démontrés, tissés de nombreuses interactions, durant différents contextes, à travers le temps et l'espace. Merci particulièrement à Guillaume Bouffard, Thomas Létan, Arnaud Fontaine et l'équipe de développement de WooKey de l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information). Merci pour tous ces échanges.

Essentiels dans le milieu informatique, peu mis en avant, merci aux développeurs d'outils de développement et d'analyse, IDEs en tout genre, et toutes les communautés derrière qui mettent à disposition leurs compétences, parfois leurs temps de loisir, pour le développement informatique mondial.

Par ailleurs, la thèse est un projet de trois ans, ce qui représente une part de vie qui a été pour ma part densifiée par la crise sanitaire avec la découverte du piano et de l'astronomie, amplifiée par le dynamisme lillois avec la danse, les sorties, les maraudes, bouleversée par les déménagements et la rencontre de cinquantaines de personnes à différents lieux de vie. L'effort de thèse s'appuie sur des fondements humains solides, qui m'ont porté, qui me soutiennent, malgré de nombreuses périodes d'absence pour délivrer ces travaux. Merci à tous mes nombreux amis, vous saurez vous reconnaître, wherever we met, wherever you are, vous qui transformez un jour ordinaire en moments délicieux. Pour cette thèse, j'aimerais particulièrement remercier Charles, fier habitant du Nord, qui m'a offert son canapé pendant plusieurs mois pour y habiter, m'a ouvert à la culture lilloise, et m'a introduit dans de nouveaux cercles amicaux dans une ville que je ne connaissais pas. Du côté de Caen, merci tout particulièrement à Valentin de m'avoir accueilli dans sa ville adoptive avec l'envie de me faire découvrir la région et merci à mon collègue de bureau, Yacine, qui grandit sa thèse en même temps que la mienne tout en partageant et dégageant beaucoup de joie malgré les journées pluvieuses normandes (non ce n'est pas vrai, il fait beau tout le temps ici, si si on me l'a dit).

Enfin, je ne suis que l'enveloppe charnue d'un noyau immuable qui me suit à chaque pas, ma famille. Certaines personnes nous ont quittés, remplacées par les souvenirs et la chaleur du coeur, mais leurs ombres éclairent ma route au quotidien de jour comme de nuit. Tack Mormor för din uppmuntran, för Mexiko resan och alla andra resorna som du hjälpte till med ! Pour terminer, merci à ma soeur Chloé, Maman et Papa de ne faiblir jamais et de m'aider à grandir, toujours et encore, avec votre amour et votre soutien indéfectibles, je vous aime.

Acronyms

SLOC Source Lines of Code. 96, 179, 230
SWaP-C Space, Weight and Power-Costs. 26, 280

T

TCB Trusted Computing Base. 15, 24, 38, 49, 50, 87, 99, 113, 286, 328

Foreword

" You will learn it later " - said a big sister to her little (too) inquisitive brother on the way to school. One day, he learned that there was no Santa Claus, but kept anyway the gifts he received, including a computer. He learned also, as the years passed, how to conduct mathematical proofs in math class and cool stuff about computers in technology class. In his leisure time, he also learned to break the passwords that restricted his allowed computer time. Unfortunately for him, security increased and there was no more material to show him the way. He faced that human knowledge had frontiers and much more uncertainties than he once thought. Nevertheless, he heard of a profession that regularly crafted new knowledge for others: researchers. That gave him hope to master his computer again one day.

Now, the boy has grown up and does not need to break passwords any longer. However, the world has also changed and the modern Santa Claus is distributing toys embedding computers that can be remotely controlled, with about the same poor security as the boy's first computer. Out in the wild, script kiddies of that time have polished their techniques and take any opportunity to annoy old classmates with much more sophisticated tools. The time has come for the big boy to take a leap of faith in the dark corners of small embedded systems security and make it harder for schoolchildren to hack their toys, if only to help them discover a profession.

Contents

List of Tables

List of Figures

Listings

General introduction

Chapter outline

Technological context

On September 15, 2022, the European Commission presented to the European Parliament and the Council a new Cyber Resilience Act [69]. The proposal raises the security standards for any digital component like software and products. Manufacturers and retailers are required to follow mandatory cyber security requirements during the whole lifecycle of their products. The CE-mark informs customers and businesses that the products followed all cyber security obligations to be on the market.

The new Act comes after a series of initiatives to strengthen the cyber security of constrained devices, among which requirements from the National Institute of Standards and Technology (NIST) in the United States of America (USA) to improve cyber security standards for Internet of Things (IoT) devices [129] , the obligation to provide software updates in the European Union (UE) [67], the ETSI EN 303 645 [68] standard from the European Telecommunications Standards Institute (ETSI) that is the first European standard for cyber security in IoT devices, the monitoring and general public dispatch of the top IoT device vulnerabilities by the Open Web Application Security Project (OWASP) [137] and a trend towards more trustworthy IoT devices [76].

IoT devices are to a great proportion composed by low-cost resource-constrained devices. Thus the questions arise: is it that such devices already include strong cyber security measures? Or must they include these measures in new products and hence rise the product prices? Or can low-cost cyber security be effective anyway and easily, immediately deployed on these devices?

Unfortunately, IoT devices already demonstrated their efficiency as harmful remote attack vectors, such as the theft of information on protected databases through a hacked smart thermometer in the fish tank of an aquarium [104], or weapons as demonstrated by security researchers by remotely controlling a car [127], by hacking a toy for children [138], by installing a ransomware on a coffee machine [121], and worse with real attacks such as the Mirai botnet [112] or the spread of an IoT chastity belt ransomware [27].

Furthermore, security is associated to the notion of confidence. There is no point to build a fortress if there is a permanent open backdoor. The ecosystem already had appetite for strong security guarantees by the use of formal verification that leverages mathematical techniques (more details in Section 2.2.4). The Ostrich algorithm does not work anymore for products that affect people's privacy and health, business and market day-to-day operations, or political and national strategical decisions.

In this dissertation, we explore means to secure resource-constrained devices with mathematical guarantees.

Work context

This thesis is an industrial and academical three-year partnership (*thèse CIFRE*, in French) supported by Orange and University of Lille. It started on November 28, 2019.

Orange [135] is the French historical telecommunications corporation. The company is interested in securing the devices connected to its networks in order to reduce malicious traffic. Hacked devices might clog the bandwidth which would in turn impact the quality of services, they might also attack Orange's own infrastructure such as routers and antennas or other more vulnerable devices to conduct massive cyber attacks profiting cyber criminals and terrorist organisations. Orange sustains this work in order to benefit the society at large and all outputs are publicly released in open-source licenses.

2XS [1] (eXtra Small, eXtra Safe) is a team part of the SEAS group [161] (*Systèmes embarqués adaptables et sécurisés*) of the CRIStAL laboratory [42] (*Centre de Recherche en Informatique, Signal et Automatique de Lille*). The team invests the field of cyber security for embedded devices. One of their recent notable contribution is the development of an operating system kernel called Pip which isolation property has been formally verified.

In this respect, the main work of this thesis has been to adapt the 2XS team's results for highly constrained embedded devices. This project is called Pip-MPU, derived from the name of the parent project Pip and from a computer component present in constrained devices that sets up protected memory regions.

Pip-MPU is also enrolled in the TinyPART project [174], a consortium of French and German academical and industrial partners, and serves as the software basis providing the isolation guarantees.

Reading guide

This dissertation is split into three parts.

The first part I deals with the thesis context.

I introduce the general security challenges in embedded devices in Chapter 1 and more particularly describe the attacker model we consider in this thesis.

Then, I present preliminary notions on operating systems with a specific focus on security, on formal verification and low-end connected embedded devices in Chapter 2.

I continue in Chapter 3 with a detailed overview of current existing security solutions that exhibit strong guarantees of isolation and find a gap concerning constrained devices.

Chapter 4 concludes the first part by introducing the thesis which motivates the design of a secure kernel for constrained devices and by presenting the contributions weaving this dissertation.

The second part II deals with the design of Pip-MPU, the adaptation of Pip for constrained devices provided with a Memory Protection Unit (MPU).

I introduce a framework to set up nested compartmentalisation in constrained devices based on the MPU in Chapter 6.

Pip is presented in Chapter 7, where it is adapted for constrained devices with MPU via the specialisation of the framework. Pip-MPU is thoroughly evaluated against Pip's security requirements and in terms of performance, memory footprint and energy consumption.

The third part III deals with Pip-MPU's formal verification of the isolation property.

The formal basis and proof intuition are described in Chapter 9.

Then, I present the formal proofs of two representative Pip-MPU services conducted in the Coq Proof Assistant [95] and analyse the proof effort. Pip's proof workflow has been leveraged and adapted for Pip-MPU.

Pip-MPU's formal verification process followed a different strategy than in the Pip project. The evolved process is presented together with leveraged proof techniques in Chapter 11.

Finally, new proof metrics are presented in Chapter 12 to monitor the proof development process and select an optimised proof path.

The second and third parts start with an introductory chapter laying down the surrounding context, respectively in Chapters 5 and 8. These parts end by a review of the proposed contributions and offer some perspectives and key takeaways.

Finally, the general conclusion in Chapter III is supplemented by material that steps back on the design, implementation and formal verification processes by giving some

insights of the followed methodologies and some intuitions, where I also engage my personal beliefs.

I provide a fast reading track recommendation in the main chapters. The recommendation highlights the important sections that the reader in a rush should read to pass technical details but still follow the main conversation.

Furthermore, I pinpoint next references to the main contributions of this thesis for details about:

- Pip-MPU: Chapter 7
- Pip-MPU's proofs: Chapter 10
- leveraged proof techniques: Chapter 11
- proof metrics: Chapter 12

The contributions of this dissertation have been, or will be, shared with the scientific community through the following scientific conferences and workshops:

- COMPAS conference 2020 [36, 53]
- RESSI conference 2020 [148, 58]
- COMPAS conference 2021 [37, 54]
- FiCloud international conference 2021 [70, 57]
- COMPAS conference 2022 [38, 55]
- IoTE international conference 2022 [97, 56] (planned in December 2022)
- Platforms and Mathematical Optimization for Secure and Resilient Future Networks Workshop [96]: presentation 'Formal Proof Metrics : the Developer's Guide to Formal Proofs' (planned in November 2022)

Furthermore, I participated to deliverables for the TinyPART project.

In addition to that, Pip-MPU and associated evaluation as well as formal verification of the isolation property are publicly released [1] [2] [3].

Finally, the work has also reached the general public by my participation to the competition *Ma thèse en 3 minutes* (translation: My thesis in 3 minutes) organised by Orange with a call for public vote [75, 134].

[1] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/master
[2] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/benchmark/benchmarks
[3] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/addMemoryBlock_proof/

I use the pronoun "I" to speak about personal contributions. "We" includes my supervisors. Authors of other referenced contributions are directly named. I use Pip-MPU to speak about the project developed in this dissertation and refer to its parent project as Pip or Pip (MMU). A detailed summary in French is in Appendix A.

Part I

Context

Chapter 1

Introduction

Chapter outline

The weak link is the primary attack vector of cyber criminals. With connected devices of all kinds operating in the IoT, cyber criminals turn to resource-constrained devices in order to perpetrate their attacks. Constrained devices (also called low-end devices) are small embedded devices that can take the shape of thermometers, light bulbs, smoke detectors, buttons... Low-end devices swarm everywhere, from our homes to our workplaces, pushed by the growth of the burgeoning market of the IoT. Consequently, sensors and actuators no longer live in isolated environments but need to be connected to respond to the intelligent ecosystems (digital twins, IoT-Cloud Continuum [77]). But in the connected world, the devices are very close to distant attackers who can infiltrate the private networks the devices are connected to.

When facing an attack, even outside the computer system world, one of the first intuitions is isolation: isolation from the attacker by taking cover or isolation of the attacker. The same goes for security issues and memory isolation is one answer to memory vulnerabilities. For example, there is high interest in isolating the interpreters, the networking stack, the file system or any untrusted third-party software from the core business logic. Isolation is *necessary* and constitutes the fundamental security principle otherwise software components are not isolated and may be overwritten, in such a way that we can't rely on their current state to infer other properties. It should be noted though that isolation is *not sufficient* as a general solution. For example, a constrained

device which upgrades its firmware with a malicious software won't be protected by isolation but through other security measures like digital signature and authentication.

Isolated components are no more vulnerable to a chained attack that would compromise the whole system and, in a more holistic view, the ecosystem by the compromission of a single element. Furthermore, isolation is the basic block of other security principles such as MILS (Multiple Independent Levels of Security) and TEEs (Trusted Execution Environments) [24, 158].

1.1 Threats and opportunities

For an operator, new devices using the networks are business opportunities. However, networks with compromised and malicious devices are market killers in that they can massively attack the networks causing resource shortage and breakdowns for legitimate users. Devices lacking security are vulnerable to all kind of attacks that can be exploited to malicious intents. It is of strategical interest to have healthy network traffics to provide quality services to customers.

For an IoT manufacturer, not presenting the new cyber security CE-mark that is introduced in the new Cyber Resilience Act [69] indirectly implies to the customers that the product does not follow the cyber security obligations and suggests it is not secure enough compared to other products having the mark. Also, a product known for vulnerability exploits causes brand reputation damage that can significantly impact the business.

Furthermore, the IoT knows no borders and the market shows that the progression rate of devices joining the IoT is at least linear [168]. Arm estimates one trillion of devices to be on the market in 2035 [166] (40 billion in 2025). Projects currently shift from legacy isolated embedded devices to connected systems. According to the 2019 Embedded Markets Study [61], 65% of the respondents tied to the embedded world will have one or more projects devoted to IoT (compared to 21% nowadays). According to the Eclipse IoT Developer Survey 2019 [71], two-thirds of the respondents are currently working on IoT projects or will be in the next 18 months. This clearly shows a movement towards connected devices. While in the IoT, security is the top developer concerns, it is ranked last in the embedded world. Even worse, 20% of the embedded devices don't have any security measures implemented at all. 26% use software-only security measures, while only 20% use hardware measures [61]. Security of IoT devices, among them constrained devices, is then crucial for market adoption with users having confidence in these devices. It also globally reduces societal and economic costs [69].

In addition to that, embedded systems usually have a long lifetime (expectations could be around 10 years for a device on battery) which also depends on their communi-

cation frequency drawing energy and in the end impacts their software updates. Their autonomy makes them good candidates to be in unreachable areas in the wild, for example launched by airplane in a forest. Due to their potentially sporadic communications, they cannot benefit from traditional IT security mechanisms like the "fail first, patch later" approach. What is loaded on these systems might work for years without much possible updates, which justify the use of formal methods to acquire as much confidence as possible at design time. This embraces the trend for security kernels and especially the security-by-design principle which suits mission-critical and high-assurance security systems' needs.

At last, we acknowledge the stringent needs for embedded devices taking the connectivity turn. Providing an open access solution would make the technology as accessible as possible. Furthermore, it would accelerate the development and deployment of products that would rapidly reach the consumer and industrial markets with minimal efforts.

1.1.1 Research questions

This thesis explores the following questions:

- Why is security so low-ranked among embedded developers and by transitivity to the systems they develop?
- How easy is it for remote attackers to overcome connected embedded systems and turn them into weapons for massive cyber attacks?
- How can constrained devices protect themselves against such attacks? Do they have the necessary technological bricks available and suited for their resource-constrained environments?
- What kind of attacks can they push back?
- What level of confidence can be expected from protected constrained devices?
- Embedded systems are known for their heterogeneity featuring different hardware platforms. Do portable solutions exist that cover the majority of these devices?

In search for answers, we focus on security isolation for low-end embedded systems (other security-related topics, such as remote attestation, are not investigated).

Isolation techniques are many-fold and intervene at each system layer from the hardware up to the applications and toolchains. They can be classified in three categories: physical isolation, hardware-based (hybrid) isolation (hardware memory protection

components, modified or not, associated with some control logic like hypervisor-OS-kernels) and software-only isolation (language-based, sandboxing and SFI, software virtual machines, hypervisor-OS-kernels without hardware support [179, 164, 170]).

In this thesis, we interest us in hardware-based memory isolation solutions. We look here at general solutions at system level, such as kernels, generic tools or modified hardware components that set up isolation measures. We leave behind bare-metal solutions that do not bring enough universality.

1.2 Attacker model

The attacker's strategy is to follow the path of the weakest link and aims to take full control of the resource-constrained device. The adversary can then modify the device to change its behavior, violate the security of other inner software components including the OS/kernel, corrupt data that is sent to decision makers, use the device to access restricted networks and attack connected computer systems or compromise many devices to launch a synchronised massive cyber attack.

We consider a powerful distant attacker who can install software components on an IoT device at reach. This could result from software initially controlled by the attacker such as a third-party software or untrusted installed application or from a primary attack which continues.

Our attacker is nevertheless not able to perform any hardware attacks, neither from a distant location [82], nor by close distance because of the required physical proximity not necessarily true in the case of low-end IoT devices and high-levels of hardware integration making them economically not attractive for an attacker [111].

This lets functional assumptions on the hardware untouched, *i.e.* the hardware is reliable by being functionally correct and fully operational. Thus physical attacks are out of scope. Side-channel attacks are also ruled out.

Software attacks are particularly harmful for the systems through the exploitation of memory vulnerabilities such as programming errors, stack overflows, illegal memory access, data corruption. The adversary tries to access memory out of its memory space (read, write, execution permissions) by exploiting memory vulnerabilities like code injection attacks, code reuse techniques, illegal read and write operations [105, 32, 111], so by a Write-What-Where vulnerability exploit [52]. The attacker could then inject arbitrary code and control data, stack, heap, owned peripherals of any installed applications, and could circumvent normal control flow by pointer or stack manipulations [153]. The attacker manages to gain full control of the device by overtaking a privileged component or be granted security critical privileges, known as a privilege escalation attack. This situation is due to flawed software because we consider initial software components

benign but containing software vulnerabilities.

Usually, software developers know the importance of testing and code analysis to identify and lower the risks of bugs. However, despite all these efforts, bugs are still found and vulnerabilities discovered frequently even in large open-source communities [43]. The risk is spread over the device's lifetime and malicious components can be installed by direct attack or by unsuspected/undetected delivery as it is common to use SOUP (Software-Of-Unknown-Provenance) and COTS (Commercial-Off-The-Shelf) software and other third-party solutions of doubtful trust, which may sometimes receive remote software updates and remote commands from unknown issuers.

In any case, memory vulnerabilities are present in our systems and can be exploited.

This scenario is realistic as reflected in the figures of the Common Vulnerabilities and Exposures (CVE) database [44] reporting a total of 144877 vulnerabilities' entries:

- 6052 entries match the keyword 'memory corruption' [48]
- 13555 entries match the keyword 'overflows' [49]
- 23723 entries match the keyword 'arbitrary code' [45]
- 1779 entries match the keyword 'kernel privileges' [47]
- 1566 entries match the keyword 'root privileges' [51]
- 2212 entries match the keyword 'privilege escalation' [50]
- 191 entries match the keyword 'elevate' [46]

These keywords are used in the reports' description such as: *"A malicious application may be able to execute arbitrary code with system privileges"*.

Chapter 2

Preliminaries

Chapter outline

This chapter draws the necessary background concerning the next two parts of this dissertation. It gives the reader some insights over the challenges that are addressed in this dissertation and understanding of state-of-the-art solutions presented next. The quick reader might want to jump to the presentation of the Memory Protection Unit (MPU)2.5.1.

2.1 Operating systems and kernel

The frontier between the Operating System (OS) and its kernel is blurred, as much as their respective definitions.

Andrew Tanenbaum defines it as 'the software that runs in kernel mode — and even that is not always true' [172]. It handles two features: hardware resource management (processor, memory, inputs/outputs,...) and abstractions for higher-level applications (drivers, files,...).

On the contrary, Richard Stallman defines the OS as 'a collection of programs that are sufficient to use the computer to do a wide variety of jobs' and the kernel as 'one of the programs in an operating system—the program that allocates the machine's resources to the other programs that are running' [149]. Richard Stallman considers then Tanenbaum's OS as a kernel.

Modern OSes also integrate the dimension of security, in addition to previously mentioned functionalities.

In this work, we are close to Tanenbaum's definition and sensible to the OS security. The next sections precisely define our view on operating systems and kernels.

2.1.1 Definitions

The **Operating System (OS)** of a computer system provides an abstraction of the hardware resources that are mainly the processor or *Central Processing Unit (CPU)*, the memory and a set of peripherals. Applications access OS services via a well-defined API (Application Programming Interface) that launches system calls. OS services could be features such as storing, networking, or displaying information on a display. Thanks to this OS abstraction, applications are portable on different platforms independently of the underlying hardware. This is like a human who would be as comfortable using the same application but on a different computer or a mobile version, because the interface is the same or similar even if deeper layers are different (*i.e.* the OS and the computer).

Note that applications do not need an OS to run, they run **bare-metal**. However, removing the OS abstraction forces applications to stick to the hardware and thus be less portable. It is also a concern of security and safety since the application takes more responsibility. If an application badly configures a peripheral or tries to access

a critical part of the memory, it could endanger the system and as a consequence the whole ecosystem it is integrated in.

The **kernel** is a subpart of an operating system which contains all the critical features. What is or not a critical feature is determined by the kernel's philosophy and that's why there are kernels varying in size and categorised in different families. More than distinguishing differences in software components, this distinction between kernel and the rest of the system can also be the basis of privilege separation [159]. Modern OSes have two or more hardware-enforced privileged modes which restrict what less privileged modes can do. The most privileged ones can disable interrupts, change system settings, switch privileged modes, and reconfigure sensible hardware components among other things. This prevents lower privileged levels to do so and potentially gives untrusted code rights for full system control and ultimately harm the system. Privileges can be hierarchised in protection rings (inner rings have more privileges than outer rings). In small embedded systems, there are typically two modes nowadays: privileged and unprivileged. In this context, when the privilege separation is associated with hardware privileged modes, we speak of kernel space (privileged) and user space or userland (unprivileged) because the kernel is generally the most privileged component of the system.

Extending that, the least privilege principle means any software component should execute with the least set of privileges required to perform its operation.

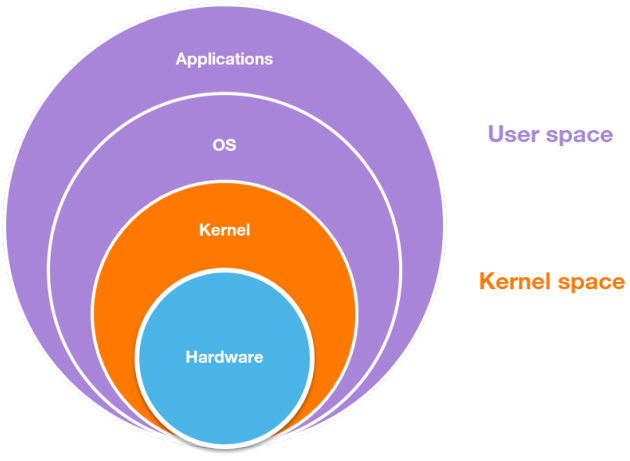


Figure 2.1: Typical scheme of hardware and software nesting. The kernel and user spaces are the software layers whereas the hardware is represented here as a general abstraction of the processor, memory and peripherals.

2.1.2 Monolithic kernels *versus* microkernels

Two main categories of OS structures/kernels are frequently opposed: monolithic kernels and microkernels. Monolithic kernels are usually used by general-purpose operating systems, like Linux [119] for UNIX-like operating systems. They concentrate all the functionalities required so to let the applications and the user interface exclusively run in user mode. The kernel itself exposes a large API that surfaces an important code base (Linux is composed by more than 36 million lines of code [133]). However, customization can reduce drastically the code base (*e.g.* MuLinux [124]) and modules can be loaded at runtime which makes it a modular kernel.

Instead, microkernels, like Minix [171], reject outside the kernel space most of the core features of the monolithic kernels. Indeed, they draw the frontier of the kernel to only keep the hardware management feature, scheduling, multiplexing and inter-process communications, but no abstractions such as file systems and drivers. Thus, they provide only a minimal set of system calls, automatically reducing the attack surface. These system calls are then used by outside servers (programs) that will reproduce the missing features of the monolithic kernels. This way, the kernel code base scales down a lot. However, due to frequent privilege switches between kernel and user spaces, performances are impacted.

2.1.3 Exokernel *versus* protokernel

Kernel types can be further trimmed down first with exokernel [64] and more drastically with protokernels [26]. The principle is to separate security from abstractions [136].

Exokernels consider it wrong to include abstractions at all in the kernel space and to make implementation choices. They are only concerned about the hardware level, for memory management and secure multiplexing. All abstractions are instead moved to the user space. Applications then choose which abstraction implementation suits best their needs. They can be linked with what is called LibOSes (Library Operating Systems) in charge of the interactions with the kernel [65].

Kernels from the protokernel family are just concerned about memory management and control flow management, only focusing on security without abstraction. Everything else, including multiplexing, must be handled outside the kernel.

2.1.4 Embedded operating systems

Operating systems for embedded systems include any form of the previously presented OS structures. However, in the early days, they usually run bare-metal, without hardware abstractions, and probably libraries like LibOSes to share features nowadays

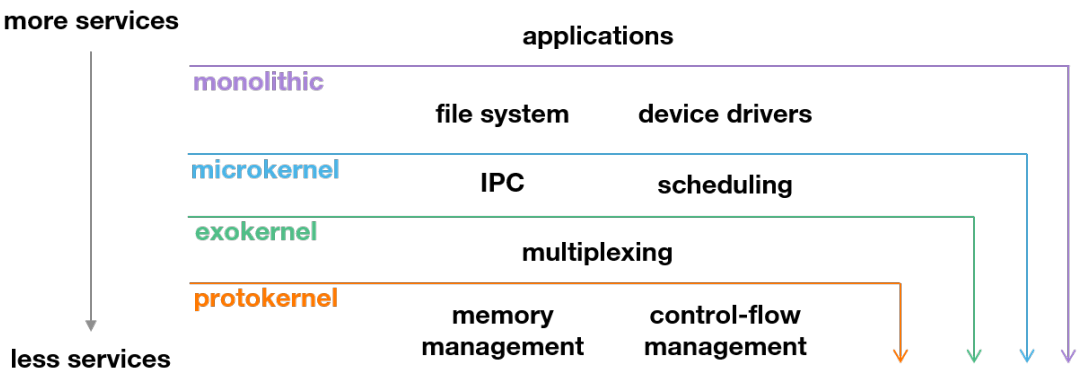


Figure 2.2: Kernel types, inspired by [100]

attributed to operating systems. There was no separation between kernel and user spaces.

Modern operating systems for embedded systems (RIOT [151], Contiki [39], Zephyr [189], FreeRTOS [73]) still do not force privilege separation or have alternatives implementations to support it.

2.2 Security

2.2.1 Cyber security, safety and dependability

Integrity and confidentiality are basic cyber security principles.

Definition 2.2.1 (Integrity)**.** Integrity means information held by some objects cannot be tampered (modified or destructed) by other objects.

For example, the author does not want this dissertation to be modified without him being aware of the modifications.

Definition 2.2.2 (Confidentiality)**.** Confidentiality means information cannot be leaked or made available to unauthorised objects.

For example, the author wants to keep his bank card PIN number private.

(Cyber) security is a very broad term and has different definitions depending on the context of use. We give our definition of security perceived all along this thesis and distinguish it from safety.

Definition 2.2.3 (Cyber security)**.** Cyber security is the capability to protect a computer system, or set of computer systems, or intangible assets, from external factors having read or write permissions, *i.e.* respectively compromising the confidentiality or the integrity.

What are authorised operations or not is defined in the *security policy* that lists the security requirements. The security policy should never be circumvented. It is distinct from the *protection mechanism* which enforces it. For example, a hardware component attributes resources across subjects in the system. Then, a security policy could state restrictions about some subjects and resources and a protection mechanism should implement the policy, *e.g.* a reference monitor, access control abstractions, cryptographic keys, *etc*...

Definition 2.2.4 (Safety)**.** The capability of a system not to harm its environment and assets (including human beings).

If security is about protecting from external factors, reliability is about protecting from internal factors, which may have been caused by security issues. Reliability is defined as the assurance of keeping the system in a safe running state despite internal events. For example, bugs and software faults should not interfere with the system's missions, so some related notions are fault-tolerance techniques (the capability to face faults) like fault-containment (faults should not propagate to other system components) and redundancy (a failing component is backed up).

We speak about dependable systems when they are at least secure and reliable. This gives the system's user a certain perception of trust. These are not the only notions to characterize a system and its users' perception of trust since a dependable system is influenced by multiple factors stemming from its inner components, outer components and environment.

All these terms are sometimes confused since they participate in the holistic view of the system and relate to other concepts such as RAMSS (Reliability, Availability, Maintainability, Security and Safety).

It is important to note though that security is a prerequisite for both reliability and safety. Indeed, an attacked system is in an uncertain state that may be different from the initial design state when the assumptions to ensure reliability and safety were set. Hence, the approach of this work is to focus on the security properties of the system.

2.2.2 Secure operating systems

Humans are not perfect and make mistakes. In the cyber world, these lead to "bugs" and unexpected behaviors of the software or hardware. But for cyber attacks, these are vulnerabilities and opportunities to harm a system.

Confidence in a software component is usually tied to its correctness. This is demonstrated by experiments, tests, simulations and code reviews. The outputs are compared

with the expected outputs given known inputs. Sufficient code coverage and absence of evident bugs give the user a sense of assurance.

High-assurance systems are systems requiring strong guarantees of some properties. Examples of such critical operational environments are the military or some health services. In these environments, system failures cause significant damage.

Moreover, from Section 2.1, we implicitly understand OS security as protecting the kernel, or other OS features, even for systems with less assurance needs. There is a supplementary consideration with security, that is to secure the applications. They can directly harm the system because of bugs or vulnerability exploits that force a crash or violate the confidentiality and integrity of the kernel or other applications. For example, in embedded operating systems where the OS and applications share the same address space, one way to secure the OS is to set up a security policy stating the separation of privileges of kernel and applications. The two security considerations are solved by setting up a security policy that embraces them both. However, there is no security policy without isolation.

2.2.3 Memory isolation

Memory isolation is a key factor in the design of an operating system or kernel. It has the purpose of well-defining the boundaries of the system components. Isolation is linked with the cyber security principles of information integrity and confidentiality. It is the basis for any security polices, otherwise there are no certainties about the system state and its properties that could be modified and eventually clash with the security policy. It naturally helps understand the global system but also builds confidence when it is enforced by the system. Security isolation lowers the risks of a vulnerability exploitation [164].

Spatial separation, or spatial memory isolation, is a type of memory isolation where data of some protected entity is correctly isolated from entities that should not access it. In other words, isolated entities should not be able to access private data or devices except their own. It is distinct from *temporal separation* that must ensure isolation of service resources like performance or latency. Spatial partitioning is a type of spatial separation by partition isolation [156].

Considering levels of security isolation, the *airgap* security is one of the most extreme applications of components' isolation, where these components are not connected in any ways with each other because located on different devices or memories for example. On the contrary, when components are just isolated by namespaces but merged together during compilation, it only offers a shallow isolation important for the design time but not guaranteed during the execution time. Memory isolation comes then with more or less guarantees and could be supported by hardware components for a higher isolation

confidence (and lower the required trust in other system components). We could export this analogy to a home environment with two houses, yours and your neighbour's. Your neighbour is particularly curious and tries to break into your house. The strongest security for the isolation of both families is to keep them separate in different houses (airgap security) so you don't need to trust your neighbour. In another scenario, you could imagine moving in together, in the same house and even share rooms. In this manner, you lower the isolation to almost nothing (really nothing or based on rules) but inversely need a high confidence in him/her (little to no security measures). Otherwise, you could merge both houses into one but still keep them separated by walls (security by employing hardware components). The thicker the walls are (the more isolation capability it shows), the less confidence you need to have in your neighbour. To extend the analogy further, we could imagine that you and your neighbour are invited to stay in a common friend's house. Your host splits you in different rooms and you need to ask permissions to do things in his home (your host is the operating system and you and your neighbour are processes). This way you are isolated from other guests and have different rights compared to your host. You need to ask your host for a service or privilege to do a certain operation.

It is important to note that isolation relates to two opposite movements: isolation from something and isolation of something, *i.e.* protect internal assets against external factors versus protect external assets from internal factors. These movements could be addressed separately by a system exposing different properties (for example, the kernel in the kernel space has more privileges than applications in the user space). As such, the properties differ in a system with heterogeneous levels of guarantees even if both movements follow the same common goal of security and safety. For high-assurance systems, isolation properties could be eventually supported by formal verification.

2.2.4 Formal verification

A software's state space is combinatorial and assurance by testing methods masks the rest of the reachable states that can cause troubles. This especially concerns security and safety because usually only functional properties are tested.

Formal methods are the only way to reach strong guarantees of the system's properties by the use of a mathematical model. Properties proven on the model via mathematical logic reflect properties in the real system. In other words, it replaces the demonstration of confidence by mathematical proofs. It is always a simplification of the real system and must be as close as possible to real behaviors.

Formal verification is a process where the goal is to demonstrate that an implementation satisfies its formal specification. The correctness of the implementation, or the

satisfiability of the specification, is demonstrated using formal methods of mathematics, like logical and mathematical proofs. It is different from validation which finds out if the specification is right in a particular context. Verification can consist in verifying invariants that are properties that should always be satisfied despite transformations of the mathematical objects they refer to. For example, a loop invariant is a logical condition that is true at the start of the loop and still true at the end of each iteration of the loop. To facilitate the proof conduct, proof developers rely on theorems and lemmas.

There are two main categories of formal verification tools: abstract interpretation and theorem provers [145].

Abstract interpretation do not need to run the program to verify the properties. It explores the state space to prove those properties are satisfied and, if not, brings counterexamples. Tools based on abstract interpretation, like model checkers, are subjects to the state explosion problem meaning they cannot explore the whole state space. The results could then be undetermined. States are then usually over-approximated, giving false positives or false negatives. They usually fail to prove high-level properties and only prove very specific properties due to the mentioned drawbacks. They are mostly automated but might have connections with theorem provers to enable human intervention. The main advantage is that they scale well to large code bases.

Theorem proving covers instead the whole state space and can prove high-level properties such as functional correctness (the implementation satisfies the specification). They largely use formal deduction. Tools are theorem provers, also called *proof assistants*. However, they are not automated, thus require intense manual guidance and expertise. Hence, formal verification of a system using a proof assistant consists in writing the specification of the properties to prove and then to prove the specification is satisfied within a theorem prover.

Proofs can be done at different abstraction levels. Higher-level proofs can capture very complex properties in a simple abstracted representation of the system. Like Edsger Dijkstra says: “The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise” [59]. From there, the abstract model can be refined down to include more details until the implementation level. The *refinement* is the traversal from specification level to implementation level by verifying complex properties on the most abstracted level and preserving them down. There can be numerous abstraction levels before reaching the binary level. Proofs can also be done directly at *implementation level*.

2.2.5 Security kernel *versus* separation kernel *versus* partitioning kernel

Many kernels striving for security exist. We review here three types of kernels. A *security kernel* is a centralised agent that controls access to system resources and gathers all security-critical software. In a security kernel, subjects (software entities) to system resources (hardware and software) must follow a system-wide security policy.

Rushby [155] introduced the notion of *separation kernel* to enhance security kernels with local security policies. Indeed, separation kernels conceptually distribute subjects on physically isolated machines, while they are in fact running on the same processor. In other words, it simulates a physically distributed environment on a single machine. The separation kernel's role is to provide the environment and communication channels between the isolated components (seen as *virtual machines* or *regimes*). The purpose is to facilitate formal verification of the kernel and each system component individually.

The *partitioning kernel* [156] appeared in the context of avionic standards for safety and is a type of separation kernel with a focus on fault containment. It allows to split software components with different criticality requirements and assesses each one of them individually. They can be assessed by *certification* which verifies the conformity to a set of standards. Certification is done by an independent party that checks the targeted standard's requirements are met. The highest certification levels, for example EAL7+ of the Common Criteria [35], require the use of formal techniques. Certification of only critical components, and not all system components, reduces the costs in the end and pushes towards a reduction of the code base. It also has the benefits to group functions usually distributed on the same machine. There is a distinction between spatial and temporal partitioning, respectively to ensure no control of a partition over another (memory or private peripherals) and ensure the quality of service for a partition is not affected in any way by another partition.

Hence, a separation kernel is a security kernel considering isolated components in the same machine and a partitioning kernel is an enriched separation kernel that focuses on safety.

2.2.6 Trusted Computing Base (TCB)

The Trusted Computing Base (TCB) is the minimal set of security-critical components (hardware and software) that needs to be trusted in a system. The TCB is compared against the rest of the system which behaviour does not affect security. In operating systems, the TCB is usually the kernel and the hardware on which it executes, but also all the protection mechanisms and may contain other security functions, and not only on the system's chip, like secure cryptoprocessors. The smaller the TCB, the better [164], or like Andrew Tanenbaum once said in a keynote I attended: «Smaller is safer» [4] (or

more informally, it is the application of the KISS[1] principle).

2.3 Legacy embedded systems and connected devices

Opposed to general-purpose systems, embedded systems are given a specific mission. They are, as the category name suggests, integrated into a larger system. They are thus less visible but very present. In a home, for example, we would typically find household appliances having embedded systems to control them: refrigerator, toaster, alarm clock, recent TVs and their remote controllers, smoke detectors... And with a modern car comes a dozen to a hundred of embedded devices for the different car native systems and advanced driver-assistance systems (ADAS). In the same home, we would find only a few general-purpose systems like computers.

In the last decades, we see more of these devices connected to other devices or exposing an interface for humans who can access to their devices remotely. By giving the devices the ability to speak with one another and with people, we created the *Internet of Things (IoT)*, a smart environment capable of sensing and interacting and serving people. While many definitions of the IoT exist, we only focus in this dissertation on the connectivity part of the IoT devices and their consequences, and essentially see IoT devices as connected embedded systems. This shift to the connected world is an accelerating trend. Indeed, according to the 2019 Embedded Markets Study [61], 65% of the respondents tied to the embedded world will have one or more projects devoted to IoT in the next year (compared to 21% nowadays). This meets the Eclipse IoT Developer Survey 2019 [71] where two-thirds of the respondents are currently working on IoT projects or will be in the next 18 months. This clearly shows a movement towards connected devices which expels the embedded world to a much higher order of magnitude [61, 71]. High economic expectations are forecasted for the first to flood into the market, but might also later become a required move to survive in this new worldwide smart environment.

However, crucial differences need to be highlighted despite the convergence brought by connectivity. Legacy embedded systems and connected devices don't have the same design curve and the same legacy, which is of outmost importance for the security concern where the figures contrast sharply: security is the top IoT developer concern [71] whereas it ranks last in the embedded world [61]. Furthermore, only 4% of the design time of an embedded system is spent on security/privacy threat/risk assessment. Nevertheless, the embedded world seems to acknowledge the need for security in their systems since security is the top 3 greatest technology challenge [61]. But the battle

[1] *Keep It Simple Stupid*, or *Keep It Safe and Simple*

for security has not come to an end: on the 25 billion of IoT devices expected on the market (2021) [76], the great majority of them will suffer from cyberattacks and the new business ecosystem will be the playground of cybercrimes capturing 300 billion to 2 trillion dollars worldwide and every year [146].

Bringing connectivity to legacy embedded systems also exposes these devices to cyberattacks as never before, which competes with the expected reliability of IoT devices; sometimes we even trust them with our lives. For example, in 2015, two researchers managed to remotely hack a Jeep Cherokee, taking control of some inboard systems and critical physical systems such as the steering and braking, foresighting tragical scenarios [127]. Moreover, IoT security is not only about attacks the way down to the devices, but also the other way around. For example, the Mirai botnet which took place in 2016 broke down major parts of the internet by taking control of a large number of devices [112]. As the IoT is scaling up, we can easily imagine the impacts of a similar botnet attack would be even more intense if it happened today. Examples in recent years are the STUXNET worm, uncovered in 2010, which was specifically designed to sabotage an Iranian nuclear facility [184] or cyber attacks on the Ukrainian power-grid in 2015 and 2016 [160, 66]. Thus, IoT devices, or connected embedded systems, have more than a local impact and the consequences should be taken from a more holistic view. With potentially high impacts, the ecosystem is in a stringent need of strong guarantees.

2.4 Low-end/constrained *versus* high-end embedded systems

2.4.1 Low-end/constrained embedded devices

A *microcontroller* is a small computer that contains a processor (or commonly Central Processing Unit, *CPU*), memory, and various input/output (I/O) peripherals. It has a limited memory and CPU speed to minimize the Space, Weight and Power-Costs (SWaP-C) factors. Despite the SWaP-C factors expected to be low, embedded systems can vary drastically in size and complexity. For example, a car might embed hundreds of high-performance microcontrollers as opposed to the low-performance microcontroller in a toaster.

In this dissertation, we will focus on this constrained type of devices, called *low-end devices* or *constrained devices*, which must deal more than others with limited resources of power and memory. Low-end devices are classified in separate categories in Table 2.1. These characteristics lead to hard constraints in the system design. In this dissertation, we are concerned about Class-2 IETF constrained devices.

Name	Data size (*e.g.*, RAM)	Code size (*e.g.*, Flash)
Class 0, C0	< 10 KiB	< 100 KiB
Class 1, C1	10 KiB	100 KiB
Class 2, C2	50 KiB	250 KiB

Table 2.1: Classes of constrained devices (KiB = 1024 bytes) [93]

2.4.2 High-end embedded devices

High-end embedded systems are still affected by the SWaP-C factors and provided with similar or fewer resources than general-purpose systems, however do not exhibit as steep resource constraints as low-end devices.

Examples are smartphones, television sets, robots or subsystems in need of powerful computation platforms like in the avionics.

2.5 Memory Protection Unit (MPU) *versus* Memory Management Unit (MMU)

Applications require memory to store their code and data, but they need to follow the kernel's security policy. This means memory has to be managed and access to resources must be controlled. The *Memory Protection Unit (MPU)* and the *Memory Management Unit (MMU)* are hardware components taking on this role.

2.5.1 The Memory Protection Unit (MPU)

The Memory Protection Unit (MPU) [15, 14] is available for small embedded systems only. It is an optional component on many ARM Cortex-M processors [13]: Cortex M3/4/7/0+/23/33/35P/55. Its role is to restrict memory accesses based on a memory layout configuration. The configuration consists in a set of memory regions called MPU regions having various permission rights (read, write, execute) and additional attributes (caching, buffering...). That is, a system usually has a default memory map that can be changed by configuring the MPU and each running process must stick to the active MPU configuration. The number of MPU regions that can be configured and protected at the same time is implementation-defined, generally 8 to 16 MPU regions. The MPU is configured at runtime by a privileged software (typically an OS kernel). A faulty memory access ends in a **memory fault**.

Systems protected by an MPU offer a higher level of security compared to their counterpart without MPU or not using it. Naturally, this goes along with a proper MPU configuration and self-protections in order to set up the protection measures as intended.

The MPU can be traced back as far as in the ARMv4t.

In the Cortex-M series, architectures ARMv6/v7/v8 support respective Protected Memory System Architectures (PMSA) as an optional extension. The MPU implements this extension to protect a 4GB address space. All micro-architectures of the Cortex-M series propose an optional MPU, except Cortex-M0 and Cortex-M1 [16]. From ARMv8-M and the introduction of the TrustZone (memory split in a secure world and a normal world), there is one optional MPU per protection domain, so one MPU for the secure world and one MPU or the normal world (or non secure world) which can implement a different number of memory regions and be existent or not. In ARMv8-M, the MPU regions are not allowed to overlap and the only restriction to set up a region is that the starting and ending addresses must be aligned to a multiple of 32 bytes. Instead, previous ARMv6/7 versions allowed MPU region overlapping and had strict size and alignment constraints: the MPU regions base addresses are aligned with the region size that is a power of 2. ARMv6/7 also provides MPU subregions, that are eight equally sized subregions that can be independently enabled for each MPU region.

Since Arm dominates the mobile, embedded and IoT markets, it means the MPU is *de facto* the most widely available hardware component to protect memory assets and already used in many OSes for constrained devices. However, the reality is different, the MPU is not present in all implementations or, when it is present, is set aside. Several reasons push device manufacturers not to use the MPU: too high memory footprint, too high energy consumption, too high performance overhead, too high time-to-market pressure to take the time to integrate, compatibility issues, limited regions and thus flexibility or no guarantee of system protection [194].

Equivalent units exist on other processor architectures, such as the Physical Memory Protection (PMP) on RISC-V [152].

2.5.2 Differences with the MMU

From a broad perspective, the MPU is for small embedded systems what the Memory Management Unit (MMU) is for general-purpose systems, since both share the memory protection role in a similar fashion. We summarize the key differences between MMU and MPU in Table 2.2.

MMUs organise the space by memory pages, usually of fixed size, instead of MPU regions of variable size. However, the MPU is much more constrained and systems with MPU support up to 16 MPU regions compared to millions of memory pages for an MMU. Hence, the MPU ensures hardware-based memory protection similar to the MMU (read-write-execute access control rights), but does not virtualise the memory. Furthermore, the MPU's configuration is stored in CPU registers while the MMU manages page tables stored in the main memory. As with MPUs, illegal access ends up in a memory fault.

Attributes	MMU	MPU
Virtual memory	Yes	No
Configuration mode	privileged	privileged
Memory region unit	page	MPU region
Number of memory region unit	Millions	8-16
Access control (RWX)	Yes	Yes
Configuration storage	main memory	registers
Device memory size	MB-GB	kB
Device frequency	GHz	MHz

Table 2.2: MMU *versus* MPU.

The highlighted differences prevent us from directly transposing from an MMU-based system to a system based on an MPU. The limited number of MPU regions, designed accordingly to the requirements of constrained devices, does not scale with the millions of pages protected by an MMU. Furthermore, they are configured and they operate so differently that the configuration software should be entirely redesigned.

Since our interest lies in constrained objects, the MPU will be our central hardware concern in the rest of the document.

Chapter 3

Present situation in constrained devices - State of the Art

Chapter outline

In this chapter, we review state-of-the-art hardware-based isolation solutions for low-end embedded systems. We do not consider performance in our study because the researched solution should first and foremost exhibit strong security guarantees. At first, we explore existing solutions that only leverage hardware components to set up memory isolation with various granularities. Then, we look at the combination of hardware-based solutions with software-only solutions to strengthen the overall isolation confidence (hardware-based solutions enhanced with language-based isolation or formal proofs, or software-only solutions hardened by hardware components).

Our study brings to light the lack of an open secure hardware-based solution for

constrained devices providing strong isolation guarantees, illustrated in Table 3.1. Hence, we review corresponding solutions in high-end embedded systems where several formally verified OS kernels are identified.

3.1 Hardware-based isolation solutions

Hardware is the foundational trust anchor in any system. Some parts are specifically crafted for security, like processor security extensions or dedicated hardware such as crypto modules, that are in charge of cryptographic operations.

Some hardware components can help with isolation. We review in this section the TrustZone technology and the MPU (more insights in [163]). They are considered the strongest forms of isolation because, if correctly configured (we will see in the problem statement section 3.3 that this might not always be true), they cannot be easily attacked by software. Note that we ruled out physical attacks, even as a consequence of a remote attack, that might disturb the correct functioning of the hardware [179]. But this can also be a disadvantage because hardware is difficult to update (it can be considered as frozen software) and thus cannot receive security patches. A long time can separate hardware generations and as much time for cyber criminals to exploit the security flaws. Hence, hardware-only solutions are not realistic long-term solutions, and hardware is rarely used without software controlling the hardware that can receive patches. For example, the MPU does not offer isolation natively, nor protection, so a certain form of logic must control it. This section highlights the kernels, architectural components and tools that manipulate security hardware for isolation.

Furthermore, this section also narrates research projects that modify the hardware to add security extensions.

3.1.1 Embedded system kernels

In the last decades, many efforts stemming from research and the industry have created a plethora of operating systems and kernels for low-end devices. The mainstream OSes in this category are RIOT OS, FreeRTOS, Zephyr, TinyOS, Contiki, MbedOS [165, 71].

They offer disparate isolation guarantees which will be reviewed in the following with other appealing OSes and kernels showing some form of isolation guarantees. Furthermore, systems using COTS security hardware might not actively provide efficient isolation mechanisms. The MPU for instance, can be used for memory protection only (*e.g.* restricted access rights for memory sections or stack canaries), which does not imply isolation.

TrustedFirmware-M [118] ARM Cortex-M devices incorporate the TrustZone [19], a hardware extension ("security extension to the embedded world") that splits the memory into a 'Secure World ' and a 'Normal world'. Trusted components and security-critical functions are supposed to be placed in the 'Secure' part and the untrusted code, such as a rich OS is supposed to run in the 'Normal' side. Code is executed in one world or the other depending on where the execution address is, such as the trusted code is completely isolated from the untrusted code by design.

ARM provides the TrustedFirmware-M which is their reference implementation of a 'Secure World' OS. In this way, TrustedFirmware-M runs in the 'Secure World' and another 'non-secured' OS runs instead in the 'Normal World' (like MbedOS, FreeRTOS and Zephyr detailed later). TrustedFirmware-M provides a Trusted Execution Environment (TEE, also called in Arm terms Secure Processing Environment or SPE), which is the root of trust (RoT) of the IoT devices with crypto, attestation, firmware update, secure storage (keys on internal memory and confidential data on external non trusted memory) security features.

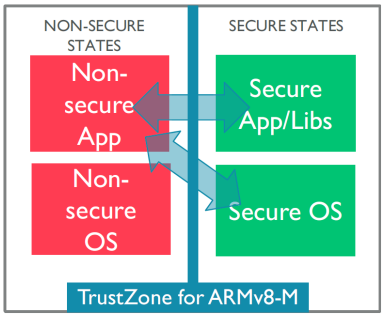


Figure 3.1: ARM TrustZone technology.

However, memory is split into two parts and can not be extended. For example, trusted code is not necessarily trustworthy and could also benefit from memory isolation between components, which is not possible to achieve alone with TrustZone. TrustZone can nevertheless be used in conjunction with optional MPUs in each world providing a limited set of clustered memory areas. The privileged components of the system manage the MPUs.

RIOT OS [22]: no isolation guarantees by default with MPU-based extension for self-stack isolation, microkernel RIOT OS has been growing in popularity in recent years and is specifically designed for low-end IoT devices. It offers numerous ports to a large panel of boards and CPU architectures. In its default mode, the RIOT application is compiled together with the RIOT OS, and no isolation is enforced at runtime.

For the devices equipped with an MPU, RIOT brings the possibility to use up to three MPU regions: one region to prevent execution in the on-chip RAM (SRAM) and two regions to set up stack guards for the active thread and the Interrupt Service Routine (ISR) respectively in Thread Mode (operational mode) and Handler Mode (interrupt and exception handling). Stack guards are defined at the bottom of the stack, such that a write in this area will result in a memory violation.

This MPU configuration is fixed at compile-time, accessible by the privileged application, and prevents buffer overflow attacks because of the guards and code injection attacks because data becomes non-executable. However, this doesn't isolate applications from each other and leaves the possibility to directly modify the OS and as such take control of the whole system.

Zephyr RTOS [189]: no isolation guarantees by default with MPU-based extension for self-stack isolation, kernel data isolation, kernel stack guards Zephyr RTOS targets connected resource-constrained devices and embedded devices, and provides a full networking stack to this aim. User threads run in user mode, while kernel and kernel threads run in supervisor (privileged) mode. When hardware memory management is supported (MPU or MMU), Zephyr offers static boot-time MPU configurations to restrict code to read-only execution, protect privileged kernel RAM, and protect privileged peripherals by separate MPU regions. At each context switch, an MPU region also protects the user thread stack.

Additional memory blocks can be given to a given thread (memory partitions) and can be shared between threads, each block requiring an extra MPU region.

There is also an extension to detect (but not protect) privileged thread stack buffer overflow by setting MPU-enabled stack guards.

If hardware memory management support is missing, Zephyr offers the option of adding canary values to detect stack overflows. However, data could be corrupted and there is a delay in the overflow detection.

In its most MPU-protected mode, Zephyr protects then against stack buffer overflows and limits the execution of code and isolates kernel data. However, the high number of static MPU regions coupled with the stack guards gives less possibility to add supplementary memory blocks (since each requires an extra MPU region, bounded by the hardware requirements). Furthermore, kernel threads run with the kernel and could interfere with it. Finally, when threads launch their own child threads, the latter inherit the same memory partitions as their parent forcing mutual trust and less flexibility.

FreeRTOS-MPU [74]: MPU-based kernel isolation, partial self-stack isolation, microkernel In its common use, FreeRTOS offers no isolation means.

However, the FreeRTOS-MPU variant makes use of the MPU and the privileged and unprivileged modes to isolate the kernel from the tasks and to restrict the tasks' scope. Indeed, two static MPU regions are requisitioned to protect kernel code and data from unprivileged tasks. Accessing the kernel regions from these tasks will result in a memory fault stopping the execution. One MPU region sets the tasks' code to read and execute only so that no task can write in its own or in another tasks' code area. One MPU region is used to set the peripherals as not executable. As peripherals are usually memory-mapped, this prevents code injection attacks by peripheral configuration. Finally, either the task is unrestricted and allows writing in the whole RAM except the kernel data, or the task is restricted to itself and can define three supplementary custom regions (which can overlap regions of other tasks). In this configuration, the kernel is protected at any time but tasks expose their data and stack to the other (potentially malicious) tasks.

Furthermore, a task can also be created as privileged and as a consequence has the same rights as the kernel, exposing this time the kernel to vulnerabilities originating from malicious or flawed applications.

EwoK [25]: MPU-based process and kernel isolation, MPU-based between process isolation, MPU-based driver isolation, microkernel, use of Ada/SPARK EwoK is a driver-oriented microkernel which isolation policy is determined at compile-time.

EwoK follows the least privilege principle: a module or kernel should not have more rights than it should. In other words, even the kernel should not directly access the modules. Indeed, the kernel is solely privileged, so could have access to the whole defined memory, but when the kernel code is executed, the MPU regions dedicated to the user modules are disabled. Hence, the MPU effectively isolates the kernel (code, data, peripherals) from the user modules at any time.

EwoK uses the ARMv7 architecture possibility of defining eight equally-sized subregions. They are enabled independently to handle drivers more finely (several subregions could be used for one driver, or each driver uses one subregion). At context switch, the subregions corresponding to the new active driver are enabled, as well as the peripherals fitting the driver. This way, the drivers are isolated from the rest.

EwoK also uses SPARK which is a formally defined programming language strengthening the Ada strongly-typed programming language by automatically generating verification conditions to remove runtime errors (buffer overflows, division by zero...) and by optionally establishing functional specification (program contracts describing the expected behaviour, pre- and postconditions...). These formal conditions are proved automatically or may require human intervention through the interface with theorem provers. In EwoK, the SPARK contracts cover some critical functionalities by specifying what the subprogram should do and enforce the $W \hat{} X$ principle.

3.1.2 Isolation generation tools

Existing applications must be ported on these systems in order to inherit the isolation mechanisms brought by the previously exposed kernels and operating systems. But this comes with additional development costs and technical challenges. Research also investigated isolation mechanisms with hardware support by using tools and toolchains that instrument the code. They do not offer the set of functionalities provided by the previously studied systems, but offer an alternative lightweight mechanism, usually with a tiny separation kernel in charge of the memory space isolation preparation. They need additional offline firmware analysis on the source code.

MINION [105]: MPU-based process isolation The MINION architecture tackles the isolation problem together with the real-time performance issue. It proposes a per-process switching mechanism that avoids frequent privileged mode switching. The memory view switcher is responsible for reconfiguring the MPU at each process switch (so a relative runtime overhead) following fixed MPU configurations computed during compilation. It is the only code running in privileged mode; the rest are unprivileged modules. They identified the reluctance of system designers to use the MPU and proposed a way to statically analyse a firmware and reorder the memory sections to group the ones of similar nature and isolate processes and kernel.

However, it does not address bare-metal systems and it is subject to vulnerabilities because of memory view over-approximation due to the hardware's limitations.

ACES [32]: MPU-based function to process-level isolation in bare-metal systems ACES (Automatic Compartments for Embedded Systems) is a binary instrumentation tool to automatically generate memory compartments. It's an extension to the LLVM compiler which goal is to optimise the code. Similar to MINION, fixed MPU configurations are computed during compilation and enforced during runtime depending on the active protected components. The great benefits are that it doesn't require to modify the source code since it leverages the compiler and the process is fully automatic once a compartmentalisation policy is chosen, making it a smooth process. For example, the tool can be used to isolate components originating from the same source file from the components in other source files, or by peripheral use, or by functionality. The compartments can be smaller than a process or thread; it just needs a bit of code and optional associated data. The developer chooses the compartmentalisation policy that best fits the usage, at the expense of performance (frequent calls to components of other compartments which adds compartment switching instructions) or security (mixing together security-critical components with untrusted third-party components). Indeed,

the ACES instrumentation stages end by lowering the number of compartments with the actual available MPU regions (eight to sixteen depending on the target board).

Thus, compartments ideally isolated may finally be mixed in the last effort to adapt to the hardware constraints.

OPEC [195]: MPU-based function to process-level isolation in bare-metal systems OPEC also instruments the binary to automatically generate isolated compartments (called *operations* and are independent tasks with their inner functions). It enforces strict memory isolation (stack isolation and global data shadowing technique). It features MPU region virtualisation with explicit MPU reconfiguration on demand during runtime to prevent the memory-view approximation of MINION and ACES. The compartments are created respecting an automatically generated policy, that statically analyses the firmware. A small separation kernel is linked to the firmware image to enforce the policy by the use of the MPU. It also emulates store and load operations on core peripherals. Compartment switching is done by instrumenting the code with additional system calls that trigger the isolation-safe context switch.

OPEC only targets bare-metal systems and requires the source code, so does not support dynamically linked libraries.

MultiZone [139, 140]: MPU-based domain isolation MultiZone is a small separation kernel for ARM Cortex-M devices.

Isolated domains are actively protected by the MPU at runtime, given a fixed configuration defined at compile-time.

It also offers a real-time scheduler and a communication sub-system in charge of inter-domain communication.

Legacy applications must be ported into the user space and previously privileged register accesses are trapped and emulated by the kernel.

3.1.3 Modified hardware components

Because of the numerous hardware limitations, some projects propose modified hardware components, optionally operated by a small software; for example by extending the CPU instructions or enhancing memory bus access logic.

While they show reasonable performance for embedded systems use cases, the required hardware customisation may be too expensive for low-end devices.

TrustLite [111]: EA-MPU-based trustlet isolation TrustLite protects trustlets, a limited number of security-critical functionalities (e-payment, attestation module), ensured by a modified MPU.

The MPU is made execution-aware, meaning it controls the execution rights of the instruction issuer in addition to monitoring the data accesses. The root cause is because trustlets and the untrusted OS and its tasks shouldn't access the resources equally, while they share the MPU configuration fixed at boot time.

The execution-aware MPU (EA-MPU) first checks the rights of the requesting entity before checking the data access rights. Rights are read-only accessible for everyone in the Trustlet Table. The architecture also embeds the Secure Loader, a software component which purpose is to load the trustlets, configure the MPU at boot, and eventually load and launch the untrusted OS and its tasks (which can be trustlets). Besides, it integrates an exception engine that saves the stack and the instruction pointers as well as general-purpose registers in the trustlet's protected data area before handling an exception. This way the OS exception handlers are out of the trustlets' TCB and prevent information leakage.

It also provides trusted IPC (Inter-Process Communications) by implementing a communication protocol between trustlets.

CheriRTOS [187]: task isolation by hardware capabilities CheriRTOS is an OS providing a great number of isolated tasks. It claims hardware mechanisms such as the MPU or the TrustZone are difficult to orchestrate and thus neglected. CheriRTOS instead uses memory capabilities, *i.e.* authorization tokens with base address, top address, and access permissions. Memory capabilities give the capability owner access to memory segments defined by the capability. The capability model is based on CHERI [181] and so CheriRTOS runs on a modified processor with capability extensions and a capability co-processor.

With CheriRTOS and the modified hardware platform, performances are better compared to using an MPU.

While the processor security extensions are not yet available for micro-controllers, Arm developed and released in 2022 the Morello board [17], a CHERI-extended processor for high-end devices. The Morello board enables fine-grained compartmentalisation [18]. If adapted for micro-controllers, and because the embedded sector extensively relies on Arm, the ecosystem might change the security model.

3.2 Mix of hardware and software solutions

Obviously, hardware-based isolation mechanisms can be combined with usually software-only solutions. The defence-in-depth concept [24, 6] acknowledges the need for several independent layers of security to assure the system's global security expectation, with isolation being one of these layers.

The thesis does not focus on software-only solutions that are not used in combination with hardware mechanisms, even the ones exhibiting intermediate to strong security guarantees like JavaCard Virtual Machine [98, 7] or Security MicroVisor and PISTIS [3, 83] leveraging Software Fault Isolation (SFI) [170, 180] that is a widespread technique to isolate programs in "logical" address spaces by code instrumentation.

Dynamic selection of isolation mechanisms is not yet available for constrained devices, like Flex-OS [113] for high-end devices. Flex-OS lets the developer choose the mechanism that fits best the desired isolation and performance goals at compile-time (SFI, hardware isolation like MMU or capabilities...). But this implies isolation guarantees vary depending on the involved mechanisms.

MbedOS with TrustedFirmware-M [20]: MPU-based memory protections, TrustZone-protected in conjunction with TrustedFirmware-M Nothing prevents systems from mixing security hardware components and relying on other hardware-based solutions, as MbedOS illustrates.

MbedOS is the IoT OS developed by Arm, freely open-source. It is a multi-threaded real-time OS for embedded devices featuring storage, connectivity, and drivers to standard peripherals, among others.

The ARMv8-M port enables TrustZone support by relying on a specialised TrustedFirmware-M implementation [118] to host the security features. In this architecture, the MPU is used to isolate partitions. It enables automatically the MPU to write-protect the ROM (Read-Only Memory, where code and defined variables lie) and to execute-lock the RAM, thereby implementing the $W \hat{} X$ principle stating that no memory regions can be at the same time writable and executable and so prevents code injection attacks.

However, these memory protections can be disabled when needed by an application and can't be extended to protect other memory regions in a fine-grained manner, which strongly limits the applicable security policy.

TockOS [175, 115]: MPU-based process and kernel isolation, MPU-based between process isolation, use of Rust TockOS offers process-grained isolation enforced by the MPU.

The MPU is dedicated to the processes and is reconfigured at each context switch to restrict the memory access rights of the current process. The MPU configurations are mostly fixed at compile-time to protect code, data, stack, and heap of the active process. Some MPU regions are used for Inter-Process Communications (IPC).

The kernel contains the core functionalities (scheduler, hardware layer and configuration) and drivers (SPI, UART, timer...). When the kernel is executed, the MPU is disabled, such that the kernel has total access to the whole memory. The kernel is

thus trusted notably because it uses the Rust programming language [157]. Rust still offers access to low-level details with the ‘unsafe‘ keyword. It has no garbage collector meaning the memory safety properties are not ensured at runtime by the programming language itself (like Java) but through the compiler's static analysis at compile-time, which makes it an efficient language suitable for embedded systems. This is to lower the risk compared to using not-safe languages like C, which have direct access to the hardware and the memory. This fits the needs for low-level programming but allows risky operations from developers.

MPU-hardened Java Card Virtual Machine [30]: virtual machine, MPU-based kernel, MPU-based context isolation This project aims to protect JavaCard applets running on the same hardware by the use of the MPU.

Indeed, the MPU is used to guard executing contexts that each has a Java Card byte code interpreter. Each applet is then confined in its own context with its interpreter. The MPU configuration is fixed at compile-time and the MPU is reconfigured to always match the executing context. The kernel has access to the entire memory and is the sole privileged program running in kernel space while the applets all run in user space.

The virtual machine also leverages the MPU for memory protection by implementing the $W \hat{} X$ principle and to detect overflows on the active applet's stack and heap.

ProvenCore-M [114, 28]: process and kernel isolation, between process isolation, TrustZone or airgap isolation of the trusted core, fully formally verified in SMART ProvenCore targets the ARM-empowered IoT industry by offering a trusted core ensuring between process isolation and kernel and process isolation by formal proofs. The core is a modified MINIX microkernel [171] and makes sure processes are isolated from each other.

The typical configuration implies setting ProvenCore aside a rich OS (like Android since they target specifically the mobile sector) and letting all critical security features like authentication or e-banking be handled from ProvenCore. Indeed, ProvenCore is meant to be placed in the 'Secure World' of the TrustZone or a separate microprocessor/microcontroller in conjunction with the rich OS in the 'Normal World'. The critical features are replaced in the rich OS by a proxy process handled from ProvenCore.

The isolation ensures the confidentiality and integrity of the protected ProvenCore processes and the formal proofs follow a refinement strategy: first, the isolation properties are proved on an abstract model considering separate machines for each process, then refining down to a concrete model used for the C code generator later compiled by CompCert [186] (a formally proven compiler). At each refined layer (four in total), a

commutation proof states the changes and checks the properties hold. They received the Common Criteria EAL7 certification, the highest possible certification 2.2.5.

The Cortex-M variant is similar and runs on Cortex-M microcontrollers with TrustZone or a supplementary microcontroller, and is the sole running OS (except the rich OS).

3.3 Problem statement

State-of-the-art solutions show different isolation techniques that exhibit various levels of guarantees, ease-of-use, and functionalities. Hybrid approaches combining hardware-based and formally proven isolation expose the strongest means of isolation. However, no solutions for constrained devices reach that level of guarantee.

Indeed, the majority does not rely on formal methods to prove isolation, which is necessary to get a mathematical guarantee of the proposed solution.

Also, many solutions restrict the number of protected components because of hardware limitations, numerous in constrained devices, which take them away from the ideal isolation solution. Isolation is often seen as a trade-off to performance with negotiable security guarantees.

In addition to that, many solutions are tied to a specific architecture, like Armv7-M.

Moreover, when used, the MPU is usually not accessible anymore to the developer, who has no choice but to conform to the fixed policy decided before runtime.

When they do use formal methods, they do not address abstract notions such as isolation at task level, do not analyse the full TCB, are also tied to a specific hardware architecture implementation, and are still very limited in the number of protected components. Many also require manual intervention of the system or application developer. This is error-prone since they are almost all written in an unsafe programming language like C and risk putting the system in a dangerous state. Some take support from properties of the programming languages to increase security by avoiding certain classes of memory vulnerabilities, however this does not address isolation from the system perspective. Indeed, hardware isolation appears stronger since it treats not the causes of the vulnerabilities but their effects.

And lastly, many current systems have no means of isolation and are vulnerable to code injection attacks, privilege escalation, data theft or corruption, and system corruption.

Though the technological bricks are available to build a hardware-based memory isolation solution for constrained devices with strong guarantees, the process requires experts in system design and formal methods. As the market has risen rapidly in recent years, the industry was more concerned with securing sales before the devices and

any investment in security implies additional costs to feed through expected low-cost devices. This difficult equation left blank the field of strong security guarantees for constrained devices as illustrated in Table 3.1. To the best of our knowledge, ProvenCore-M is the only isolation solution to target the microcontroller sector and to expose strong guarantees of isolation by formal proofs. The major drawback is its proprietary classification, which is why few inner workings have been disclosed.

Finally, from the literature review arising from our field of research, there is no solution which:

- is **software-defined** AND
- provides **strong isolation guarantees** by being at least
  - **hardware-based** AND
  - ensured by **formal proofs** AND
- fits **constrained devices** AND
- is **open-source**.

3.4 Hardware-based solutions in high-end embedded systems

In contrast to previous solutions, we focus now on high-end embedded systems. High-end embedded systems have similar memory isolation hardware mechanisms to common general-purpose systems, such as the MMU, which are well-established security mechanisms.

We first review hardware-based solutions with intermediate isolation guarantees and follow by studying solutions exhibiting strong isolation guarantees.

3.4.1 Hardware-based solutions with intermediate isolation guarantees

High-end embedded devices range from systems with a few MBs to several GBs that are more like general-purpose systems. For this latter class, devices typically support mainstream OSes like Linux distributions, Windows, MacOS and Android [183, 126, 8, 78]. For the rest, more specific embedded solutions exist with equivalent environments like members of the Windows IoT family, Linux Embedded, and Fuchsia [125, 62, 79].

Virtual memory provided by the MMU is always used for high-end embedded devices, and so they all provide isolation by using the MMU to stop the system in case of a memory isolation violation (the famous memory management 'blue' screen in Windows reproduced in Figure 3.2).

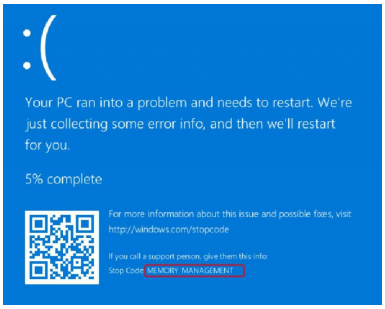


Figure 3.2: Windows 10 stepping into a memory management error.

The MMU is not the sole isolation hardware component, for example TrustedFirmware-A [117] based on the TrustZone, equivalent to TrustedFirmware-M for high-end ARM Cortex-A devices.

3.4.2 Hardware-based solutions with strong isolation guarantees

Many research efforts focused on verifying the correctness and security of general-purpose systems, embedded systems, and separation kernels [192, 193, 106]. We review here three fully formally verified OS kernels from the last twenty years, in addition to the ProvenCore kernel already covered previously with its micro-controller variant ProvenCore-M [28].

seL4 [109, 173, 89] : MMU-based process and kernel isolation, MMU-based process isolation, fully formally verified in HOL

In 2004, a team from NICTA presented seL4, the first ever fully formally verified microkernel. This means there are mathematical proofs that the kernel implementation follows its high-level specification that encompasses isolation properties.

seL4 is a member of the L4 microkernel family [116] which provides four major features: virtual memory management, threading, scheduling and inter-process communication. seL4 also features capabilities for authorisation which associate any kernel object with a set of access rights, limiting the behaviour of the object.

The guarantees provided by the formal verification are rooted in the hardware making it a kernel with very strong means of isolation (notably the MMU, but not the software capabilities). The complete verification in the Isabelle/HOL [177] theorem prover could only happen because of the few features it provides. The proof follows a refinement strategy, conducting verification of system and security properties on high-level specifications and then refining down to the implementation level.

As a microkernel, seL4 can be the secure foundation of full operating systems

featuring more system components, like the recently announced KataOS [81] (2022) almost entirely written in Rust.

mCertiKOS-secure [41, 86, 84]: MMU-based process and kernel isolation, MMU-based process isolation, separation kernel, fully formally verified in Coq

mCertiKOS [86] is a single-core processor kernel that derivates from the CertiKOS kernel which also used refinement (almost 40 abstraction layers) to fully formally verify the system's functional correctness [84]. A variant, mCertiKOS-emb kernel, stems from mCertiKOS and is specifically crafted for embedded systems by removing virtual memory support and user-space interrupt handling. mCertiKOS kernel and variants allow authorised communication channels between different user-space applications by message passing.

In 2016, mCertiKOS-secure [41] is designed as a pure separation kernel by modifying the mCertiKOS kernel. In the modified kernel, inter-process communications are disabled. mCertiKOS-secure proves confidentiality (non-interference) by a high-level security verification on each system call. Then, different levels of abstraction are linked by simulation that preserves the security property down to the low-level assembly execution. The kernel is fully formalised and verified in the Coq proof assistant [95, 40].

Pip [101, 99, 100, 26]: MMU-based process and kernel isolation, MMU-based process isolation, separation kernel, fully formally verified in Coq

Created in 2016, Pip is designed and implemented as a member of the protokernel family 2.1.3, drastically lowering the features to the sole memory management and context switching features.

Pip is a separation kernel where partition isolation is formally proven and hardware-based on the MMU, in a different way than proposed by seL4 or mCertiKOS. Pip's formal proofs are directly conducted on the source code, which corresponds to the lowest specification levels in seL4, ProvenCore, or mCertiKOS so without refinement. The small exposed feature set makes it very flexible for upper implementations, so that designers could keep their actual project while soaking the system with strong isolation guarantees.

	Intermediate isolation guarantees	**High** isolation guarantees (supported by formal methods)
Low-end embedded devices	TrustedFirmware-M [118], RIOT OS (MPU) [22], Zephyr RTOS (MPU) [189], FreeRTOS-MPU [74], EwoK [25], MINION [105], ACES [32], OPEC [195], MultiZone [139], TrustLite [111], CheriRTOS [187], MbedOS [20], TockOS [175], hardened Java Card Virtual Machine [30]	ProvenCore-M [28] (proprietary)
High-end embedded devices	Windows IoT [125], Linux Embedded [62], Fuchsia [79], TrustedFirmware-A	seL4 [89], mCertiKOS-secure [41], Pip [26, 99], ProvenCore (proprietary) [28]

Table 3.1: Hardware-based solutions (OSes/kernels, tools, architectures) classified by isolation guarantees for embedded and general-purpose systems.

Chapter 4

Design a secure kernel for constrained devices

Chapter outline

Academia and industry witnessed major advances in high-assurance systems in the last twenty years through the development of fully formally verified kernels (seL4, CertiKOS, Pip). However, these systems target high-performance computers and are not suitable for low-end devices because of an absence of technology or economic incentives.

4.1 Thesis

The open-answer questions asked at the beginning of our dissertation 1.1.1, our initial study exhibiting plentiful isolation solutions and even strong isolation guarantees achieved for high-end devices based on hardware components which have equivalent protection roles in constrained devices 3, the industrial and consumer benefits for security in embedded systems helped by recent regulatory obligations 4, the formal verification field maturity [91], allow us to formulate the following thesis statement:

Thesis: Constrained devices, especially connected ones, gain from greater security which is available with current technological bricks and compatible with their expected functionalities in a dynamic ecosystem. A hardware-based security solution, immediately deployable on these devices, adapted for real-world use cases, and exhibiting strong guarantees of security, can benefit consumers and the industry.

The goal is to never let an application interfere with other applications or the operating system in a way that would endanger the confidentiality and integrity of protected software components or allow be granted full control of the system; for example by exploiting a memory vulnerability.

A kernel that provides flexible strict spatial memory isolation 2.2.1 lets devices freely evolve in a dynamic ecosystem and can prevent a remote attacker to take full control over the device and perpetrate further attacks in the network. Formal verification of the isolation increases user confidence in the system and contributes to the adoption of IoT ecosystems, which in turn directly benefits the industry.

4.2 Thesis approach

This thesis narrates the path taken to secure constrained devices with immediate, effective, scalable effects.

Our approach to secure constrained objects is to design a formally-verified OS kernel providing hardware-based flexible but strict isolation by a COTS Memory Protection Unit (MPU) 2.5.1).

Three possibilities arose: 1) build a new formally verified kernel for constrained devices from scratch with oriented design for isolation and verification from start 2) formalise an existing MPU-based isolation-oriented kernel and formally prove its isolation properties, 3) adapt an existing formally verified kernel from high-end to low-end devices.

We choose to follow the third proposed approach, which is to adapt the Pip protokernel 3.4.2 to the constrained environment.

Indeed, such a secure hybrid solution fits constrained devices with minimal efforts to keep costs as low as possible.

First, our research team developed and formally verified Pip. The team has the system and formal verification expertise to conduct the adaptation.

Second, isolation is a basic security principle that serves security (confidentiality and integrity to prevent tampering or leakage of information 2.2) but is also useful for other domains by echo effect such as safety (fine-grained access control to critical functions, attack detection,... which increase robustness for example), information hiding, fault

isolation, subsystem recovery, software modularity, improving system structure and easing system evolution [179]. Without isolation, no properties can hold because the base on which the properties hold is subject to change. It is the first security barrier when setting up a defence-in-depth strategy [169, 6] that is the principle to set up multiple security layers to counter cyber attacks. Pip's flexible partitioning scheme makes it possible to design secure and flexible solutions for constrained devices. Pip also has a minimal TCB convenient for security by reducing the attack surface, but also participates to keep adaptation costs low.

Third, Pip provides strong isolation guarantees by formal verification with an adaptable proof workflow that already showed successful results with Pip. Starting from an already formally verified kernel propagates the isolation properties and might share similar formalisation and proofs. The full software TCB is assessed so that upper layers (*e.g* OSes, applications) know which guarantees or confidence they have in their executing environments.

Fourth, Pip avoids the struggle of application verification since the partition is free to evolve as it wishes within the (MMU or adapted MPU) hardware harness controlled by Pip. Furthermore, two formal verification strategies are usually used for separation kernels, either refinement from high-level specification to low-level implementation levels or direct formal verification at implementation level [192]. Pip follows the second proof strategy and so is not concerned by the refinement paradox (avoided in seL4 and CertiKOS) [41].

Fifth, this approach leverages access control dedicated hardware by the MPU available for Arm architectures to set up Pip's flexible partitioning scheme. It is a reasonable choice because Arm dominates the market of IoT devices and legacy embedded devices thanks to its power efficiency (67% of the market shares thanks to the ARMv7 architecture [71]). Such a solution gives user applications the possibility to leverage the hardware security primitives of their chosen platforms (and not only relying on the one-level isolation of the TrustZone, which would also be compatible with this kernel). Furthermore, it does not require additional custom tools or embedded software to instrument the code or verify the firmware when loaded on the device, which could heavily downgrade performance. With Pip there is only a one-time cost during the design and implementation. Pip's cost of flexibility is compensated by the lack of flexibility in current systems.

Sixth, a COTS MPU does not require hardware modifications that would imply additional design costs. It also prevents incompatibility with currently deployed hardware platforms. Pip does not need an extended Instruction Set Architecture (ISA) as well. By using widely available hardware, the adapted kernel can be used for COTS systems thus keeping production costs low.

Seventh, Pip is open-source so security claims can be assessed by anyone.

Eighth, Pip is based on C and the Coq Proof Assistant which have widespread use in different communities. The programming languages are then convenient for low-level system developers and formal verification experts.

Ninth, the first approach incurs higher design considerations and would weaken the psychological acceptability with yet another OS kernel without proven applicability nor user community (Pip communicates through the Pip User Club [142]). The second approach, in addition to the above-identified challenges, would mean distance with the system developers, no formal verification-oriented design (Pip has been developed with a system-proof co-design approach [99]), and usually not portable solutions with a limited number of protected domains compared to what is proposed with Pip. Also, an adapted MPU-based Pip would be complementary to some enclaves, for example implemented with TrustZone-enabled devices because MPUs might be present in both secure and non-secure worlds.

In the end, this approach enables a secure-by-design kernel immediately at the disposal of the industry, enabling a minimal hardware-rooted TCB. Isolation for security provides a first layer of defence against the remote attacker considered in the attacker model 1.2 and as a consequence would prevent the attacker to gain total control of the system by memory vulnerability exploits. It gives system and application developers strong guarantees of isolation for software components running on the same resource-constrained device. It covers mixed-criticality environments and enables multiple stakeholders to securely run on the same device, with multi-process and multi-threaded components for bare-metal and OS-based applications.

4.2.1 Challenges and knowledge gaps

The solution the thesis aims to provide is surrounded by several knowledge gaps we here identify, based on our state-of-the-art study. In other words, these are the gaps that should be tackled to fulfil the goal of the thesis.

First, while existing solutions adapted their system to other architectures, no isolation kernel adapted their MMU-based system to an MPU-based system. The degree of reusability of the system design and proofs is unknown. By experience, proof efforts are reported very high in all formally verified kernels [100, 41, 109] and difficult to combine with the available time frame for the project that also involves the system design.

Second, there is no system with formal proofs of the isolation by the use of an MPU. This means the MPU is not present as assumptions of existing isolation proofs. The MPU should then be modeled and its behaviour specified to use it in the isolation proofs we develop.

Third, isolation in constrained devices without isolation proofs are tightly coupled to the hardware while we strive for a generic solution. We closely follow the memory protection hardware trends to be able to fit current hardware and its future evolution.

Fourth, the security costs, *e.g.* the altered performances, are always a source of anxiety for embedded developers. The design focus should always challenge real-time requirements.

Overcoming these challenges are the thesis' conditions of success and this dissertation's guidelines.

4.3 Results

This dissertation consolidates the thesis 4.1 by declining the approach in two major parts.

The first part deals with the design of a secure kernel for constrained devices, named Pip-MPU, which is an adaptation of the Pip kernel for devices provided with an MPU.

The kernel, Pip-MPU, is presented in Chapter 7. The design fulfils Pip's security policy and requirements. The kernel is implemented on an ARMv7-M (ARMv7 Cortex-M architecture) empowered board with MPU. The prototype is evaluated to measure the performance, memory footprint, and energy consumption costs of the solution against security gains. Pip-MPU and its evaluation are presented in the papers [55, 56] and are publicly released[1][2].

Pip-MPU is a specialisation of a framework, presented in Chapter 6, which sets up nested compartmentalisation by the use of the MPU. The framework enables the local creation of memory sub-spaces at any level of abstraction. A unique privileged entity is in charge of controlling the MPU. The framework can be specialised with different security policies, like Pip's. The framework and design rationales are presented in the papers [54, 57].

In the second part, we demonstrate that the kernel is secure and can be trusted by the use of formal methods. It deals with the formal verification of the memory isolation of Pip-MPU using the Coq Proof Assistant [95] and proof metrics orienting the proof efforts.

The first intuitions to drive the formal verification are discussed in Chapter 9. The proof methodology is presented and provides a preliminary informal proof of Pip's security properties on two representative services. In order to conduct rigorous mathematical proofs, the chapter also presents the formalisation in Coq of the isolation

[1] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/master
[2] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/benchmark/benchmarks

invariant, services and low-level primitives, and hardware model. The model connects with equivalent structures and functions in the C language that are used to produce the final executable of Pip-MPU. Step-by-step details of the key instants of the formal proof of Pip's security properties using Coq are presented in Chapter 10. The proof development is evaluated using common proof metrics for formally verified kernels. Pip-MPU's proofs are publicly released[3].

The formal verification process leverages novel proof techniques which are described in Chapter 11. Pip's formalism has evolved to ease some parts of the proof, as have the proof techniques to lead the formal verification path. The proof development at system level is steered by novel proof metrics and a novel global proof strategy presented in Chapter 12. These proof metrics will be announced in the presentation 'Formal Proof Metrics : the Developer's Guide to Formal Proofs' [96].

Subsidiary results are perspectives on the design and formal verification processes.

The results confirm the thesis by demonstrating the design and implementation of a formally verified kernel for constrained devices.

[3] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/addMemoryBlock_proof/proof

Part II

Security for constrained devices: a flexible and portable memory isolation solution based on the Memory Protection Unit (MPU)

Introduction to Part II

In this part, we present Pip-MPU, a kernel specifically designed for setting up isolated memory spaces on constrained devices. To our knowledge, Pip-MPU is more flexible and portable than any state-of-the-art solution. In contrast to the other solutions, Pip-MPU sets up strict memory isolation which does not depend on hardware constraints or functional requirements.

Pip-MPU leverages the *Memory Protection Unit (MPU)* 2.5.1 to define and protect isolated memory spaces. Its design is an adaptation of the Pip kernel 7.2 for constrained devices following its philosophy and methodology, hence its name.

We first study and classify existing MPU-based memory isolation solutions based on their isolation and flexibility characteristics. We then propose a framework for nested compartmentalisation based on the MPU that enhances both flexibility and security compared to the state of the art. Based on this framework, we adapt the Pip kernel into Pip-MPU, a strict memory isolation solution for constrained devices offering flexible compartmentalisation based on the MPU. Then, we evaluate Pip-MPU by assessing the performances, memory footprint, and security gains through quantitative metrics. Lastly, we conclude that a rich OS can be secure-by-design by taking advantage of the isolation provided by Pip-MPU, which becomes a security kernel.

Chapter 5

Context

Chapter outline

5.1 Motivations

Constrained devices are often seen as dumb sensors and actuators, serving one or a limited number of functions. They act as servants for more powerful devices running complex applications. However, nowadays constrained devices are equivalent to computers from the 60s [182] and are even more powerful. For example, ARM Cortex-M powered devices are 32-bit processors and could have Mb-Flash memory and several kB-RAM -though very constrained compared to supercomputers- with modules allowing access control (*e.g.* MPU, see section 2.5.1) as well as peripherals allowing high-speed data transfer (*e.g.* DMA), networking (*e.g.* BLE), sensoring (*e.g.* temperature monitoring) or actuating (*e.g.* water valves). The ecosystem around constrained devices also shows increasing complexity with several available OSes (*cf.* 3) and their applications. Hence, low-end/constrained devices 2.4.1 have the necessary resources to allow complex applications. But complex does not need to mean an obscure implementation, and for developers, a platform difficult to use.

In this chapter, we aim to give more flexibility for developers to develop the full potential of their chosen hardware platforms while building reliable systems inscribed

in the secure-by-design paradigm.

5.2 Flexibility in MPU-based isolation solutions

This thesis focuses on isolation using the MPU as a first security mechanism in a constrained device. As presented in Chapter 3, the research community already invested many efforts in MPU-based security solutions which take multiple forms: standalone MPU solutions or supported by customised tools, kernels, code instrumentation, and hardware modifications. In this section, we study the security characteristics of the same MPU-based solutions. We differentiate them based on their runtime dynamism and flexibility (categorised in Table 5.1), hardware configuration, and exposed limitations due to design choices.

5.2.1 Flexibility

Static systems

Static systems refer to solutions that only configure the MPU once and for all, like TrustLite [111]. The MPU offers a fixed segmentation throughout the device's lifetime. These systems benefit from a global protection mechanism set up at bootime. All protected domains share the same MPU configuration, thus limited by the number of MPU regions. Their static nature is a major drawback in a dynamic environment like the IoT. Upgrades, like access rights changes, service additions, or loading disparate applications requiring a different memory layout, are restricted and do not satisfy our search for flexibility.

Dynamic systems

Dynamic systems benefit from a changing MPU configuration, modified during runtime, instead of a fixed global MPU configuration. The MPU is dedicated to protecting one software component at a time, usually reconfigured each time there is a context switch between protected domains. Each protected domain has the full MPU available for itself and can then protect inner components at a fine grain (process or function level). While the MPU is reconfigured during runtime, the system's permission model can be either immutable (generally fixed at compile-time) or mutable (changed during runtime). Only a few systems also allow memory extension to their protection domains.

For dynamic systems with immutable permission models, the MPU configuration is fixed for each software component before boot time. However, the MPU is constantly reconfigured when a new protected domain is loaded, like a new process. Most of state-of-the-art systems fall into this category like µVisor, MINION, MultiZone, ACES, OPEC,

OS/kernel/ hypervisor/tool	*Dynamism*			*Flexibility*	
	MPU reconfiguration	Permission model	Extendable memory	Compartmentalisation nature	Nesting
TrustLite [111]	✗	immutable	✗	process	✗
Hardened JCVM [30]	✓	immutable	✗	process	✗
EwoK [25]	✓	immutable	✗	process	✗
µVisor	✓	immutable	✗	thread	✗
ACES [32]	✓	immutable	✗	generic	✗
OPEC [195]	✓	immutable	✗	generic	✗
TockOS [175]	✓	immutable	(✓)	process	✗
Mbed [20]	✓	(mutable)	(✓)	thread	✗
RIOT (MPU) [22]	✓	(mutable)	(✓)	thread	✗
Zephyr (MPU) [189]	✓	(mutable)	(✓)	thread	✗
FreeRTOS-MPU [74]	✓	(mutable)	(✓)	task	✗
Pip-MPU	✓	mutable	✓	generic	✓

Table 5.1: Comparison of compartmentalisation features in MPU-based systems. Pip-MPU is introduced in Chapter 7 as a specialisation of the framework presented in the next Chapter 6.

EwoK, the hardened Java Card VM, TockOS [11, 105, 141, 32, 195, 25, 30, 175]. They allow no user configuration of the protected memory spaces. Their inherent differences on the isolation level, guarantees and enforced security policies, are translated in as many different ways to configure the MPU.

For dynamic systems with mutable permission models, the MPU configuration is not decided at boot time yet. There, the MPU is used for memory protection but not for isolation. For instance, Mbed [20], RIOT (MPU) [22], FreeRTOS-MPU [74] and Zephyr (MPU) [189] apply some memory protections (stack guards, $W \hat{} X$ principle...) and offer some runtime defined regions that user applications or threads can configure. However, this is only a partial user configuration because of the limited number of user regions available. Furthermore, this feature can cause serious security issues if badly configured by the user.

For the vast majority of dynamic systems, software components are given a fixed protection domain size that cannot be extended at runtime. Exception is made with TockOS that can dynamically allocate a bit of a process' memory by enabling subregions. These are initially disabled when the process is initialised. Of course, previously discussed customisable regions in Mbed, RIOT (MPU), FreeRTOS-MPU and Zephyr (MPU) can be seen as memory extensions in addition to access permission changes, but are limited in number (without considering complete reconfiguration because they have the privileges to do so).

5.2.2 Portability

Existing solutions answer to specific use cases and are tightly coupled with the hardware platform. This makes them less portable. Solution designers needed to be creative to use

the available hardware, by inventing original use of the same hardware or by modifying it.

Original MPU use

The MPU architecture dictates the MPU features and its usage.

Two systems based on the same MPU architecture may not use the MPU features and constraints in the same manner and for the same purposes. For example, the ARMv7 architecture integrates subregions of equal size and independently enabled, and allows MPU region overlapping. This allows many combination for systems heavily relying on subregions for their protection domains. We take here the examples of TockOS and EwoK, but it extends to all proposed solutions. TockOs and EwoK both reconfigure the MPU at context switch to match the current process or application. However, TockOS loads one process at a time in its MPU configuration and uses the subregions for a process' inner workings, like expanding its memory or granting access to peripherals. In EwoK, on the contrary, several applications coexist at a given moment within the MPU configuration. The active application is discriminated by enabling associated subregions while the remaining disabled subregions contain the other applications. Hence, in TockOS, subregions of an MPU region are entirely used for a unique process, whereas in EwoK, subregions of an MPU region are split between several applications.

Furthermore, hardware constraints differ depending on the architecture, even for MPUs stemming from the same designer. The ARMv7 version embeds constraints like the multiple of the region size alignment and that the region size must be a power of two. The most recent ARMv8 architecture releases these constraints and can be seen as a generalisation. Nevertheless, no existing solutions use the ARMv8 version, and all proposed solutions heavily rely on ARMv7's subregion mechanism, making them useless for ARMv8 CPU-powered devices.

Hardware modifications

To strengthen security, hybrid approaches propose to modify the MPU. TrustLite, on which TyTAN is based, introduces the *Execution-Aware* MPU. This enriched MPU works as an MPU but includes the relation between the executing code and the data it manipulates, meaning it controls the execution rights of the instruction issuer in addition to monitoring the data accesses. ARM MPUs consider the rights separately. Apart from the fact that the TrustLite-based systems are static, the required hardware customisation may be too expensive for low-end devices and reduces portability.

5.2.3 Conclusion

Small embedded systems propose to establish isolation using the MPU, acting as the hardware root of trust. They often take advantage of privilege separation (kernel and user modes) to manage the MPU configuration of the protected domains.

The MPU hardware constraints limit the memory protection mechanisms available to system designers. The trade-off between hardware constraints, security, and performance has often to be assessed manually even with some automated parts in the design pipeline. Some systems had to rework their idealised protection to fit the limited number of MPU regions scaling down the final number of protection domains, which may lead to less security.

All the presented solutions above are designed for the ARMv7 architecture (except MbedOS), showing more MPU constraints than the ARMv8 version, even though some of them were created at a time the ARMv8 architecture was already released (post-November 2015) and boards supporting the architecture already existed (first ARMv8 Cortex M33 by Nordic in 2018). Nowadays, in the early 2020s, ARMv7-CPU empowered devices are still in the market and many mentioned solutions are still maintained, showing no trend of change. Nevertheless, RISC-V's equivalent unit, the Physical Memory Protection (PMP), is more flexible and close to the inner workings of the ARMv8 MPU version, making many of the solutions not transferable between these architectures. TockOS supports PMP after redesigning MPU memory isolation.

For others systems, no trade-off with security is allowed and the MPU is just enough to protect what is craved by their use case. However, any use case requiring more than the number of MPU subregions is not qualified for the protection offered by this kernel. This also shows how dependent some systems are on the underlying hardware.

Furthermore, most of these systems provide protection at some level of inner abstraction, like a process or thread, and are not protecting heterogeneous components.

Finally, some systems chose to modify the MPU which makes them less portable.

Each system tailored its memory protection at best for a specific use case, which is translated into different uses of the MPU. This means we cannot directly reuse one of the operating systems nor their memory protection mechanisms for another use case without deeply modifying them. If we chose to do so anyway, this newly adapted system would suffer from the same inconveniences of the current solutions: a specialised solution for few use cases, not deeply customisable and not reusable directly for other systems that are not similar, requiring users to understand concepts specific to the use case, as well as asking them to understand how the MPU is used and what protection it really offers, likewise the eventual hardware modifications we had to set up, and possibly trade-offs we could have made to stick to the hardware limitations and finally what choice made us eventually drop part of the idealised security solution by grouping

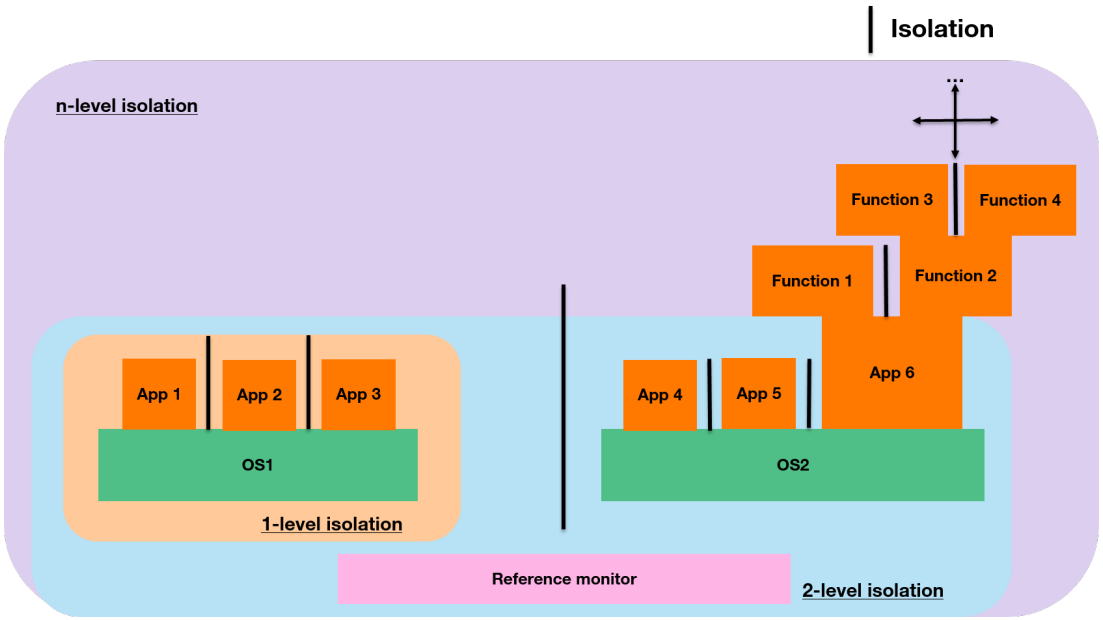


Figure 5.1: Example of different levels of isolation. The 1-level isolation typically corresponds to the isolation set up by an OS to isolate applications or processes. The 2-level isolation considers another abstraction level, for example using ARM TrustZone. The n-level isolation do not consider any hardware or software limits on the number of abstraction levels.

isolated domains together.

We do not include any bare-metal systems in our state-of-the-art review as they are tied to their platform while we consider wider applicable solutions.

Chapter 6

Framework for secure flexible nested memory spaces using the MPU

Chapter outline

The aim of this chapter is to introduce a framework to build a secure flexible system using the MPU on single-core systems. As a first step, we conduct an abstract reasoning to extract the characteristics of a flexible system. We demonstrate a nested memory space scheme is flexible and link this scheme with simplified core MPU features. Then, we propose an architecture composed of a centralised memory management entity that sets up the defined flexibility using the MPU. At last, we propose an Application Programming Interface (API) and needed metadata structures for the memory management entity. This framework and its implementation guidelines have been presented in the paper [57].

Fast reading path: The main components of the framework are exposed in Section 6.2.3 which describes the overall architecture, in Section 6.2.5 which presents the API, while the metadata structures on which it relies are provided in Section 6.3.1.

6.1 Motivations

State-of-the-art systems allow a flat isolation of protected domains, with at most two levels of abstraction (*e.g.* an OS and its isolated tasks, or an application and its threads). There is no way to configure the MPU differently beyond the attribution scheme defined by the underlying operating system or instrumentation. That is, the MPU configuration is pre-defined with at least some reserved MPU regions for a specific use.

However, the user might need more freedom and an alternative solution that allows to customise the platform on which her application is running.

6.2 Definitions

6.2.1 Secure flexible systems

Flexible systems remove the hardware constraints of existing solutions and give users total freedom of configuration, as in a bare-metal setting. The user can create and manage as many memory spaces she wants and can reconfigure each of them at any moment during runtime. However, doing so puts the system at risk with users being careless about security. As our goal is security, we add a hierarchy in the software components' relations, naturally extending common security practices (least privilege principle, protection rings, relations between OS and processes or between a hypervisor and guest OSes, ...). Software components higher in the hierarchy have more privileges than lower components (access rights and configuration). In order to keep the hierarchy, a lower component cannot increase its privileges by itself, for instance by switching to stronger rights.

As such, we give users total freedom, but *locally*. Software components are not required anymore to follow an externally defined policy, such as a pre-defined MPU configuration, as long as they follow the local policy, defined by a higher component. For example, an OS launches a process with read-write-execute rights for a sensor measurement, and the process decides to run the measurement function only with read-execute rights by reconfiguring the MPU with these rights.

In a certain manner, the privilege transfer by nested memory space creation resembles the provenance validity and capability monotonicity protection properties of the CHERI protection model [10, 9].

Definition 6.2.1 (Flexible system)**.** A **flexible system** is a set of software components that can autonomously evolve during runtime (on demand) and are related through a non-strict order relationship on privileges.

6.2.2 Flexible nested memory space scheme

A scheme of autonomous **nested memory spaces** (*i.e.* several layers of **subspaces**) fits our definition of a flexible system. Indeed, each nested memory space has less or equal memory than the initial memory space (so less privileges). For example, an OS owning a process' memory can create a nested memory space for the process strictly containing its memory. Typically, we consider the set of all unprivileged components nested in all privileged components.

Note that the system can start with several initial memory spaces which can be broken up into distinct nested memory spaces. For example, consider a flying drone with a first initial memory space completely open for user customisation to experiment flying programs, while a second initial memory space contains an emergency landing procedure. If the first memory space crashes, the second memory space kicks in (for instance by hardware) and contains memory blocks with very restricted privileges on peripherals, completely isolated tasks, and limited access permission rights. The user would still be able to resketch the memory space but starting with the restricted local permission model. Hence, several global security policies with different security requirements could coexist in the same system.

6.2.3 Security architecture for nested memory spaces

We define the architecture for a secure flexible nested memory space scheme.

Ubiquitous MPU control

Nested memory spaces are primarily management and control of memory spaces. The MPU can take over this role by defining accessible memory regions over the active

memory space. The flexible system allows any software component to design its inner memory space at runtime and in particular to create and manage a nested memory space. In the same manner, the created nested memory space holds the same ability to create subspaces. Hence, any software component has the power to control the MPU and can resketch its memory space on demand. We refer to the relation between the memory space and its nested memory space as a parent-child relationship. A nested memory space has a unique parent, whereas a parent might have several children.

Centralised MPU configuration via system calls

The MPU is part of the privileged Instruction Set Architecture (ISA) which means the configuration should be done while in a privileged mode. For software entities to reconfigure the MPU on demand, it is not an option to give this level of privilege to all of them (that include not trusted components) because of security concerns. Indeed, these software entities could attack each other by granting themselves access to other entities' memory spaces. Thus, we propose to give the exclusive role of configuring the MPU to a single entity lying in the (privileged) kernel space. The other software components run in userland. The latter should access the MPU and change the MPU configuration via system calls to this privileged entity. Due to the high privilege and potential to harm software entities if malicious or incorrectly configures the MPU, the centralised privileged entity must be trusted.

In current dynamic systems, the customisable MPU reconfiguration is at most partially possible, but mostly tied to the scheduler by loading the MPU configuration associated with the current context. This proposition goes beyond these systems, letting the user decide for each component how the MPU should be configured, dynamically at runtime, and does not assume any reserved registers for a particular use or tied with specific permission rights.

Customised security policy

Each software component can implement its local permission model on its nested memory spaces. For example, that could be enforcing the $W \hat{} X$ principle, temporary isolated memory spaces, least privilege principle, kernel self-protection, or just allowing shared memory. However, the components should still be restricted in their use of the MPU reconfiguration. Otherwise, malicious components could attack other components by expanding their own memory and by giving themselves more access rights than originally given. This means there should be a global restriction associated with local permissions. Our proposition is then not only locally flexible, but at the same time globally restricted by a security policy. For example, one component could locally create

a subspace and give it some code and data for processing, while respecting a global strict rule to not change any memory access permission rights to the given data. The global security policy is then reflected in each memory space definition and embedded within the implementation of the API.

In current systems, this global security policy would be reflected in their valid static configurations and pre-defined use of the MPU registers. However, we let the components decide for themselves how they organise their memory to free them from the frozen MPU segmentations (reserved MPU regions for code, data, peripherals...).

An example using this approach is detailed in the next chapter with the implementation of Pip's security policy.

6.2.4 Leveraging the MPU features and overcoming hardware challenges with MPU virtualisation

We explain in this section how the MPU features are leveraged to set up nested memory spaces while overcoming the strict hardware constraints of the MPU.

The memory spaces are composed of a set of memory blocks. **Memory blocks** are contiguous pieces of memory with a start address and size (or end address). Within a memory space, memory blocks might have different **access permission rights**. Memory blocks have equivalent configured MPU regions. When a memory space is enabled, the corresponding MPU regions are configured in the MPU and access is permitted. In this way, the MPU guards the memory space definition.

Nested memory spaces are subsets of the initial memory space, *i.e.* parts of the memory blocks forming a consistent whole. To extract these parts from a given memory space, we introduce a cut ability to the user, illustrated in Figure 6.1. **Cutting** splits a memory block into two subblocks, with the location of the cut address being respectively the end address and the start address of the two new subblocks. The same happens with the corresponding MPU region, becoming two MPU regions which are reconfigured in the MPU while the initial MPU region is unloaded. At this point, the user has exactly the same access as before the cut. The inverse operation is **merging** two subblocks back into the initial before-the-cut block.

However, if too many blocks are shared or created in a memory space, they will not fit in the MPU that can typically hold 8 to 16 blocks. Existing solutions deal with the hardware limitations by building a limited number of abstraction levels and pre-defined MPU configurations. In the opposite, for the user of our framework, there should be sufficient latitude to nest memory spaces independently from the effective number of MPU regions. To overcome the MPU limitations on the number of MPU regions, we introduce a **virtual** MPU. Indeed, we distinguish the full set of memory blocks composing a memory space (the *virtual* MPU) from the actual **active** set of memory

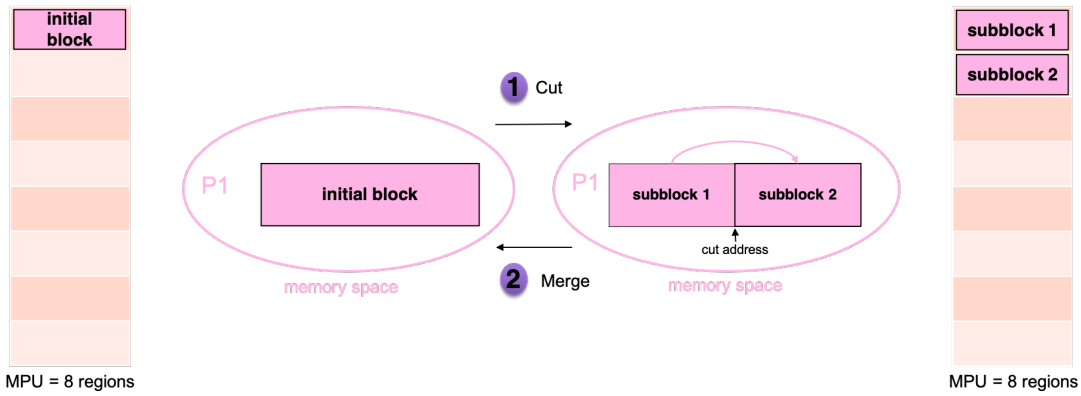


Figure 6.1: Example of a cut (1) at an arbitrary *cut address* followed by a merge (2) within the same memory space P1. The memory space keeps track of the cut by linking the two subblocks (pink arrow). The figure shows the MPU is reconfigured each time there is a mutation in the memory space.

blocks configured in the MPU. In such way, we introduce **MPU virtualisation** and the **real** MPU acts like a cache for the *virtual* MPU holding all the memory blocks of the memory space. Any *virtual* memory blocks can be elected at some point in time to be configured in the real MPU. The user is back to the reality of the hardware, only able to deal with the available number of MPU regions. The rationale behind this choice is that we consider 8 to 16 active MPU regions enough to hold the memory blocks a typical application would need (say for code, data and some peripherals). The user can decide to activate some memory blocks, while letting other blocks outside the real MPU configuration, like the fragmented memory blocks remaining from cut operations.

6.2.5 The security architecture's API

Software components interact with the centralised privileged MPU configuration entity through system calls (services). The system calls offer the flexibility of managing nested memory spaces. They are non-preemptible in order to complete the service in a consistent state. The API is both dynamic and flexible.

Dynamic API

Reconfiguration of the MPU at runtime is already present in most dynamic solutions. It usually protects a specific process and the MPU is reconfigured at context switch. This overcomes the limitation on the number of MPU regions by reusing the same regions with a different configuration depending on the active process instead of making all processes coexist at the same time in the MPU configuration. Only a subset of a memory space's memory blocks, the active memory blocks, are configured in the MPU to deal

with the MPU region number limitation.

However, a memory space *mutates* during runtime as decided by the user. We already introduced the cut ability that breaks the memory blocks composing the memory space in multiple pieces. The user can also **share**, respectively **unshare**, memory blocks with nested memory spaces to extend, respectively shrink, the available memory. **Memory sharing** is important for a process running short of memory, to give temporary access to a shared peripheral or to convey a message. Note that the nested memory space will only contain the subregions that have been shared, not the whole initial memory blocks they stem from. Furthermore, to stick to the privilege hierarchy, the shared memory block's access permissions should not exceed the ones from the original block.

Memory spaces should then be extendable (increase the number of memory blocks) and modifiable (change the characteristics of size and access permissions of memory blocks). Nevertheless, we need to register the memory space mutations (*i.e.* updates), so when a memory space receives new memory blocks or when one of its memory blocks is cut into two subblocks. The mutations are registered in **metadata structures**. User-selected memory blocks hold these metadata structures. However, for security reasons, we do not let the user access their content. Otherwise, a user could bypass the MPU configuration and access unmapped memory. Memory blocks containing metadata structures must then be disabled in the MPU and inaccessible from any software entities.

The API integrates seven system calls for dynamism:

- `add, remove`: these system calls extend, respectively restrict, the memory space of a nested software component. They can be used to move around memory chunks especially to sandbox some code or convey messages.

- `prepare, collect`: these memory space management system calls select and set up a memory block to hold metadata structures, respectively erase the content of the selected block. They are supposed to be called respectively, before adding memory and after removing memory, to split the act of extending memory from the management operations that make the registration possible. Their content is not accessible by the user.

- `cut, merge`: memory blocks are expected to grow and shrink as required by the application. Cutting and merging blocks let a software component redefine its attributed memory blocks.

- `activate`: the system call selects one memory block of the *virtual* MPU to be configured in the MPU. The user chooses the MPU region, discarding the previously configured memory block which is disabled.

Flexible API

The user is free to modify the memory spaces during runtime and so is not required to know how many levels of nested memory spaces at compile-time or boot-time.

The API integrates two system calls to enhance flexibility by creating nested memory spaces:

- `create, delete`: these system calls respectively, create and destroy, a nested memory space. The creation consists in declaring the new nested memory space. The destruction of this nested memory space sets back any shared memory blocks into the current memory space. They both use selected memory blocks to contain metadata structures concerning the definition of the nested memory space. In that respect, the content of these blocks are not accessible to the user.

This enables to break the flat memory model and, to the best of our knowledge, goes beyond the MPU configuration use in current solutions that only enables an arbitrary number of static layers (usually one to three) and cloned memory attribution schemes.

6.2.6 Summary

This section unveiled a framework for flexible MPU-based systems. The framework is composed of a security architecture and an API that outreach current solutions regarding flexibility. Indeed, they all have pre-defined limited number of abstraction levels, while the framework is user-customisable during runtime. Our security architecture is composed of a central entity in charge of memory management through the MPU. The entity runs in privileged mode while the memory spaces the user can access run in unprivileged mode. A hierarchy between software components dictates their privileges locally. A local permission model is set up by any software component when setting up a nested memory space, recursively. The whole creates as many implicit privileged levels. This means the security policy is made generic and internally defined by each component whereas current solutions usually impose a fixed memory segmentation with a security policy externally defined. However, the implementation of the system calls imposes a global security policy over the local permission models (for example, globally enforcing the $W \hat{} X$ principle). In other words, any component can implement its own local permission model *within the frame* of the implemented global security policy. The system designer decides how much freedom the software components have.

The abilities to cut and merge memory blocks are specific to systems that don't have pre-defined chunks of memory, like in the case of MPU-based systems. Specific system calls are in charge of managing the list extension or shrinkage.

The user is able to perform all operations without being bothered by the MPU constraints and does not need to follow a pre-defined MPU region attribution. This

marks a significant difference with solutions choosing to combine MPU regions in a last effort to stick to the hardware constraint with a trade-off made on security, while we strive to be as close as possible to the ideal security solution.

In the next section, we detail how the MPU features are leveraged to implement this API.

6.3 Technical implementation guidelines of the framework

We previously discussed the ability of the MPU to protect the memory space definition by associating each memory block to an MPU region. We exposed the necessity to hold a separate list of all the active memory blocks in a *virtual* MPU to deal with the MPU region number limitation. We also mentioned mandatory metadata structures registering the nested memory spaces. In this section, we present the content and the role of the metadata structures as well as answers to technical challenges when using the MPU.

6.3.1 Metadata structures

Metadata structures intervene in all system calls to register the memory space mutations.

Virtual MPU structure (or memory blocks structure)

The *virtual* MPU structure registers all the memory blocks of a memory space. A memory block gathers all the information needed to configure an MPU region. It is composed of a start address, an end address, access rights for that particular memory block, a present flag stating the block contains block information and not the default ones, and an accessible flag informing that the block can be selected for the list of active blocks of the *real* MPU. If the present flag is **not** set, the block with the default information indicates a **free slot** (available). A structure composed of only free slots is empty. All free slots in a memory space are chained in a list called **free slots list** with the first element being the first free slot. The structure is chained with other *virtual* MPU structures relating to the same memory space. The full linked list of *virtual* MPU structures gathers all memory blocks of a memory space. One can extend the number of blocks in a memory space by adding a new structure to the chained list with the `prepare` system call, and contrariwise shrink them with the `collect` system call if the structures are empty.

Sharing structure

The sharing structure holds the information on the nested memory spaces. For each memory block registered in the *virtual* MPU structure that is shared with nested memory

space (a child), it stores the identifier of this nested memory space and the location of the memory block in the main structure of the nested memory space.

The sharing structure shows the hierarchy between all software components and their parent-child relationships.

Similarly to the *virtual* MPU structure, the sharing structure is composed of multiple linked structures that can be extended, respectively shrunk, with the `prepare`, respectively `collect`, system calls.

Subblocks structure

The subblocks structure traces all the cut operations. For each memory block of the *virtual* MPU, it registers the relations between two subblocks split by a cut. Subblocks stemming from the same initial block are linked. Gathering all subblocks restores the initial memory block before the cut. Similarly to the *virtual* MPU structure, the sharing structure is composed of multiple linked structures that can be extended, respectively shrunk, with the `prepare`, respectively `collect`, system calls.

Main structure

The main structure identifies a memory space (initial or nested). The structure's location in memory represents its identifier. The structure stores its parent identifier and the location of the *virtual* MPU, the sharing and the subblocks structures. It also contains the **list of the active memory blocks**, which is used to configure the *real* MPU at each modification of the list or at each context switch. Furthermore, it stores information relating to the number of free slots, the number of metadata structures used for that memory space, and the location of the first free slot that is the head of the free slots list (chained list). As the *virtual* MPU structure has a limited number of slots, the number of free slots is used to check if a memory block can still be registered, otherwise system calls requiring additional slots return with an error. The number of structures and the reference to the first free slot are required to keep a bounded complexity as discussed in Section 6.3.2. The main structure is initialised/erased with the `create` and `delete` system calls.

6.3.2 Performance considerations of the system calls

Common to all operations is the procedure to find a block that consists in spanning over the *virtual* MPU structure. Finding a block has then a complexity of $O(n)$, n being the number of elements in the structure. However, we accelerated the procedure by the fact that the memory block's identifier is the address in the structure. Then, we just need to span over the linked list of structures until the identifier matches the start address of

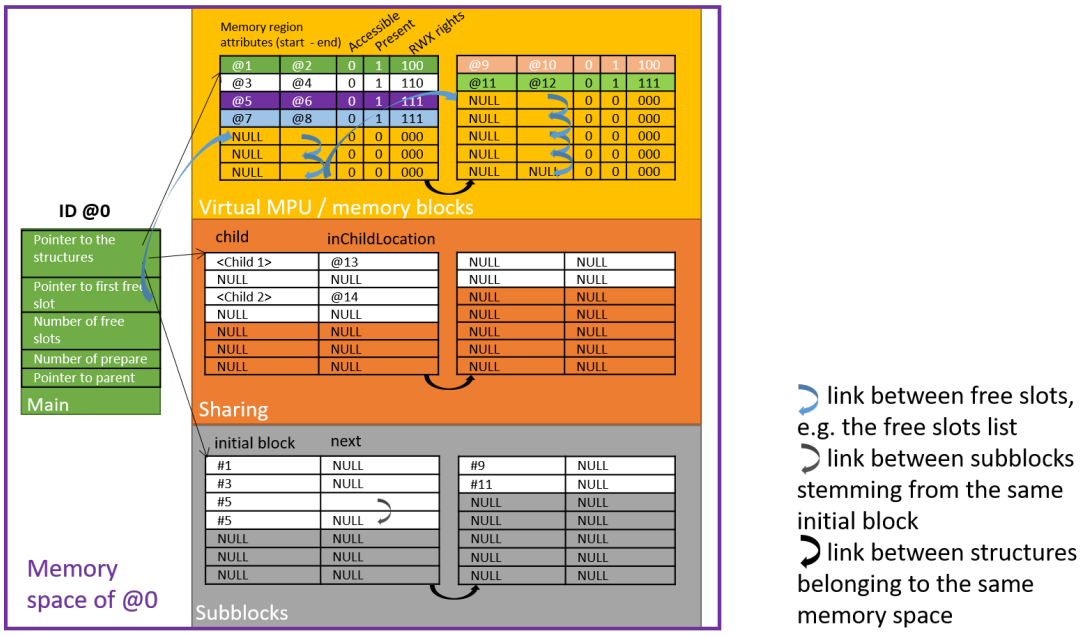


Figure 6.2: Relations between and within the structures for the illustrated example memory space @0. The chained blue arrows form the free slots list. The chained grey arrows form the subblocks stemming from the same initial memory block. The chained black arrows form the link between the multiple structures belonging to the same memory space. In particular, memory blocks @1 and @5 are shared respectively with the children 1 and 2, while blocks @3 and @7 are not shared. Furthermore, block @5 has been cut to form the additional subblock @7 as stated in the subblocks structure.

the structure. The final complexity is then $O(s)$, with s the length of the linked list of each structure in a memory space. n and s are used in the following subsections.

`create` and `delete`

The creation consists in transforming a memory block into a *main* structure. But as it should be disabled from any software components 6.2.5, the block must be discarded from all memory spaces in the system. For each partition, we must find the block to disable, so in $O(s)$ as explained above. The overall complexity is then in $O(s * p)$, with p being the number of partitions. The deletion procedure consists in setting back the state before the creation, and thus is also in $O(s * p)$ by ranging over all memory spaces. Creation and deletion don't need to be fast operations as they are not used frequently (sometimes only ones for each nested memory space at reset for static systems) and the complexities are deemed reasonable.

`add` and `remove`

Adding/sharing a block, and reverse operations merely consist in cloning/removing block information in a nested memory space. Since we target single-core platforms, block copies in different memory spaces are never accessed simultaneously. Sharing may need fast execution when conveying messages or setting up a shared memory space. The add and remove procedures are thus thought to be as fast as possible with an $O(1)$ complexity.

For sharing, the block needs to be copied in the nested memory space, and there should be a free slot where to make the copy. First, the service must find the block to copy, in $O(s)$. Having the reference to the first free slot in the main structure gives direct access to an empty slot, instead of going through the whole list of memory blocks to find one. When a free slot is used for an operation, it is pulled out of the list of free slots and the next free slot is pointed in the memory space for a future operation. The location of the used slot in the child is registered in the sharing structure of the parent. Retrieving the free slot this way and making the copy perform with $O(s)$ complexity, slightly worse than the ideal $O(1)$.

For unsharing, the block needs to be retrieved from the nested memory space. Again, it must first identify the block in $O(s)$. As it was copied into the first free slot automatically, the user has no clue where it lies in memory. However, the sharing structure holds the location of the memory block in the nested memory space. Thus, retrieving the block copy is immediate when knowing the location of the original block, which the user knows. The freed slot is inserted at the head of the free slot list where it was taken from the beginning. All operations are then in $O(s)$ complexity.

Hence, memory transfers and memory sharing do not copy data around but solely rely on memory space mutations and MPU reconfiguration which are faster operations.

`prepare` and `collect`

In order to have fast sharing and unsharing, the management operations are detached in the prepare and collect procedures. These can be called whenever required and especially way in advance so to not impact the fast sharing and unsharing procedures.

As seen above, structures are chained within the same memory space. The `prepare` operation adds new structures at the start of the linked list. In addition to that, in order to always have a fresh free slot, all free slots in the memory blocks list are chained at creation of the list with `prepare`. The `prepare` operation requires to initialise every element in the new *prepared* structures. It also needs to globally disable the chosen memory block because it becomes a metadata structure, like for the `create` and `delete` operations. The operation requires to span over potentially in all partitions to disable the memory block, because it must span over all ancestors and their descendants where the block has been shared, hence a $O(s * p)$ complexity, with p the number of partitions.

The reverse operation with the `collect` procedure scans the elements in the *virtual* MPU structure to verify the emptiness of the structure. `collect` is also responsible to set back the memory space to the state before the previous `prepare` by adjusting the information in the main structure and by correctly rectifying the references in the linked list of structures. Finally, `collect` sets back the memory block that held the structure globally, hence an overall complexity of $O(s * p)$.

`cut` and `merge`

Cutting a block splits it into two subblocks of size determined by the location of the cut. Cutting memory blocks merely consists in cloning the cut block in the same memory space, adjusting the start and end addresses and linking the two subblocks. They can be seen as `add` and `remove` operations in the same memory space, without consequences in the nested memory space, and so also have a complexity of $O(s)$.

However, a cut subblock is expected to be shared in nested memory spaces (otherwise no cut would be needed). Eventually, it can be transformed into a metadata structure in this nested memory space or in another level down the lineage, and would become inaccessible to all memory spaces. This would clash with the accessible flag set in the ancestors. Thus, as soon as they are cut, the blocks are globally set inaccessible (ancestors still have the possibility to take the memory block back by deleting the child). This requires to scan the whole *virtual* MPU structures of all partitions to find the shared block, to set it inaccessible once found and remove it from the active list of memory

System call	Complexity
create/delete	$O(s*p)$
cut/merge	$O(s*p)$
add/remove	$O(s)$
prepare/collect	$O(s*p)$
activate	$O(1)$

Table 6.1: Complexities of the system calls. s is the number of structures in a memory space (expected <=8) and p is the number of partitions.

blocks if necessary. The overall complexity is then $O(s*p)$, with p being the number of partitions. For each subsequent cut, the complexity would be in the expected $O(1)$ since the block has already been disabled globally. The same complexity exists for the merge operation to enable the block again in all partitions.

`activate`

The user decides to activate a specific memory block by identifying an MPU region that should protect it. The operation consists of two phases: 1) replacing the identified MPU region by reconfiguring the real MPU with the memory block's characteristics 2) replacing the corresponding entry in the active memory blocks list by this memory block. The length of the active memory blocks list is fixed and small (8-16, as the number of MPU regions) and the MPU is accessed in $O(1)$, therefore, we have an overall complexity of $O(1)$ for this system call.

Overview

Our study of the complexity of the complexities shows a constant complexity for one system call, a linear complexity for the sharing/unsharing operations and quadratic complexities for the rest, as reported in Table 6.1.

However, we expect a linked list of structures with a length of less than eight in implementations of the framework. In the worst case, and given a typical expected number of eight to sixteen entries in each structure, this would represent 8*16 = 128 memory blocks in each memory space, way enough to support a use case with blocks dedicated for code, data and peripherals. Hence, the computed quadratic complexities are in fact bounded by the linear complexity $O(p)$.

6.3.3 Memory fault handler

The use of the MPU implies memory space protection and the consequences of illegal access. When the MPU detects forbidden data or instruction accesses, it triggers a

memory fault that is immediately caught by the system, which in turn initiates the memory fault handler (see section MPU presentation2.5.1). The framework's memory fault handler stops the current execution of the component and reports illegal access to the parent component. The parent decides how to deal with this fault (it could be because of insufficient memory in such case the parent could share more memory with its child or because the child bugs or became rogue and tries to access memory it should not have access to).

6.4 Discussion

6.4.1 Representation of the list of memory blocks

We considered first to link the memory blocks in a sorted tree structure in order to speed up the retrieval of blocks in the system calls. But this would have encapsulated each block in a wrapped structure containing the pointers to chain the tree together, thereby requiring memory taken from the block which needs to be protected (usually a minimum MPU region size of 32 bytes) and heavier processing to organise the structure.

We also considered to split enabled and disabled blocks in separated lists, instead of packing them together in the *virtual* MPU list. But having split lists implies a block transfer between the lists, as they become enabled or inaccessible, which we are glad to avoid.

Furthermore, we could have chosen to remove the access to a memory block in a parent partition upon sharing. Thus, the memory space would need less enabled blocks and there is no need for more enabled blocks than MPU regions. However, there would still subsist the temporary states when the current memory space forges blocks by the cut operation. So if the memory space already uses all MPU regions, it would still need additional regions to host the cut subblocks, getting back to the problem of dealing with extra blocks than available in the MPU. And we consider block sharing by default essential for performance, so blocks are still enabled by default when sharing.

6.4.2 Performance

We showed reasonable expected performance regarding system call complexity. However, as the `cut` and `merge` operate in the current memory space only, they could be expected to be fast, while they do not show an $O(1)$ complexity. Nevertheless, the number of nested levels is not expected to be higher than 3 to 5 and the number of elements in a memory space is bounded, which sets the upper bound. As use cases develop, we could accelerate the operations and reduce the complexity by adding new metadata in the sharing structure that would hold the references to the original cloned

block in the parent.

Furthermore, all system calls depend on user inputs that could be invalid. An additional checking phase adds up some processing and penalizes the overall complexity for both sharing and unsharing to $O(n)$, where n is the number of memory blocks in a memory space, upper bounded with N the maximum allowed number of blocks. We acknowledge the overall complexity could be critical for use cases requiring high speed data transfer between memory spaces.

6.4.3 Automated MPU configuration

Initially, our framework [57] considered automated management of the *virtual* MPU memory blocks. It consisted in reconfiguring the MPU each time a memory block of the *virtual* MPU was accessed but not active. Our memory fault handler would search for the memory block, reconfigure the MPU with this block by selecting out another memory block in the list of active memory blocks, and redo the faulted access operation. Of course, if the access was illegitimate, *i.e.* not present in any of the *virtual*'s MPU memory blocks, the system would invoke the memory fault handler described above. This solution would release the user from picking up the memory blocks to be active and would totally abstract the MPU hardware. The procedure is similarly found in a later paper presenting OPEC [195], which virtualises the MPU, but only those dedicated to peripherals.

However, the presented framework also described an MPU region user selection option, which is now preferred. Indeed, the automated process forces us to choose an algorithm to select the active memory block to remove in order to host the faulted legitimate block (for example, round-robin as proposed). In addition to that, because it invokes a random replacement strategy, the selection algorithm could be called in inefficient moments. The developer knows best which MPU configuration is the most efficient at some point in the system's lifetime and would match as closely as possible the perfect cache policy.

6.4.4 MPU use

We have exposed how we make use of the MPU features to implement the framework.

Disabled blocks

Other uses of the MPU are possible to protect the disabled blocks (metadata structures or the initial blocks of cut blocks in the ancestors).

First, these blocks could stay enabled in the MPU but with modified permissions for privileged accesses only. Because all software components, except the centralised

privileged MPU entity, lie in user space, they would lose access to these regions.

Second, the MPU can be configured with the background region enabled, which is the case in our implementation. The background region roughly gives a privileged access to all memory not configured in enabled MPU regions. We retained the solution to pull out disabled blocks from the MPU so that the centralised privileged MPU entity can always execute without any other operation. However, we could have directly disabled the corresponding MPU regions in the MPU. But that would have polluted the MPU configuration with disabled blocks while MPU regions are scarce resources.

Third, we could have chosen not to use the background region. Without the background region enabled, the centralised privileged MPU entity would not be able to execute if not configured in the MPU, implying sharing the MPU configuration with the unprivileged components (all memory spaces in userland) and reconfiguring the MPU at each processor mode switch (kernel/user space switch). In such a case, performance would be reduced drastically and implies fine-grained management to not "forget" privileged memory when activating an unprivileged module.

Fourth, we could also have disabled the MPU when kernel code executes, as in TockOS. However, keeping the MPU always enabled frees us from the risks of not enabling it when it should be.

Fifth, we could also have inverted the procedure with a memory totally accessible from the user space and placed MPU regions with no access rights on memory portions that should not be accessed. However, given the fragmented memory layout for a complex application, there would be more regions to disable than available MPU regions and would require more complex design, maintenance and probably ease the strict global memory isolation policy which is not desired.

MPU version

In addition to that, our framework and proposed implementation do not depend on the MPU version. Indeed, we base the framework on a simplified view of the MPU, only using its core features such as the framework is designed for and encapsulates both ARMv7 and ARMv8 MPU programmer models, but is extendable to similar hardware components like RISC-V Physical Memory Protection (PMP). Many of the studied systems did not have at development time the most recent ARMv8 version enhanced with the TrustZone and more MPU regions, which releases some of the hardware constraints of the previous MPU version.

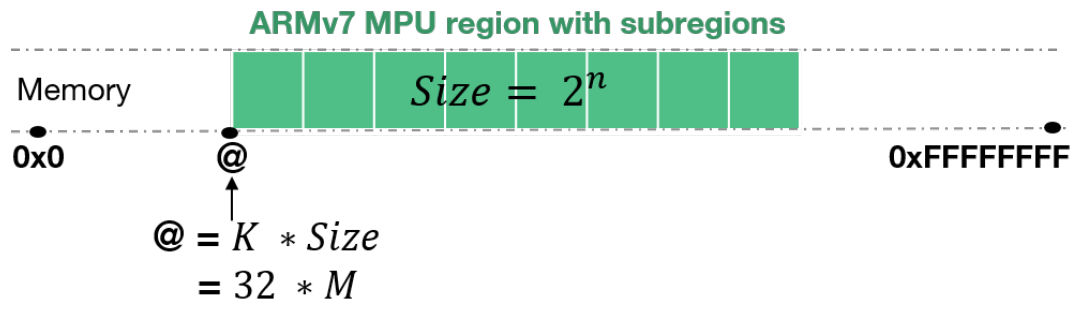


Figure 6.3: MPU region alignment constraints. The MPU region size in bytes is a power-of-two, with $4 < n < 32$ so a minimum region size of 32 bytes. The MPU region's base address is aligned on a multiple of the region's size, with $K, M \in \mathbb{N}$ such as $0 < @ < 4GB$. There are eight equally-sized MPU subregions in one MPU region, if the feature is enabled, that can be in turn independently enabled.

MPU constraints

More than that, MPU regions have a minimal size but are also subject to additional constraints like the memory region size and alignments constraints in ARMv7/v6-M MPUs. Even though the framework is based on a simplified view of the MPU and thus compatible with different MPU versions, it must deal with these MPU constraints. Concerning the MPU size, we reject any trial to craft a memory block with a size below the minimum size during a cut operation. About ARMv7/v6-M MPU region alignment, the requirements are that the MPU region must be a power of two aligned on a multiple of its size, as illustrated in Figure 6.3. The developer can align a memory space's blocks properly to satisfy the constraints, but must always deal with blocks intended for a nested memory space. When cutting subblocks, they might become unaligned depending on the cut address. In the meantime, unaligned memory blocks cause memory faults because they do not satisfy the MPU constraints.

We propose a technique to partially reconfigure MPU regions with attributes satisfying the constraints. With partial reconfiguration, a memory block is configured exactly or smaller than its normal size, but always well aligned. However, memory accesses within the part that is not configured in the MPU (the leftover to satisfy the alignment constraints) would then trigger memory faults. These faults are first processed in the memory fault handler to check if that is a legitimate access on a partially configured block, and if this is the case, partially reconfigure the MPU region with the leftover part. The latter is again partially configured to satisfy the MPU constraints. Finally, the handler replays the legitimate faulted operation with the new MPU configuration. The process is completely transparent to the user. Thus, there is a penalty for fragmented

temporary memory blocks during cut operations or badly crafted ones, while there are no penalties for final blocks that have been properly configured.

MPU reserved regions

The framework does not reserve any kind of MPU regions for a specific use, in the opposite of the majority of existing systems. The user has total control over the active memory blocks in the MPU. There are two exceptions, because of the stack and because of MPU hardware constraints.

The first exception is made for ARMv6/v7-M devices because of the constraints discussed above. Indeed, the partial reconfiguration solves the alignment and size constraints without considering the contents of the configured MPU regions. This means an unaligned *virtual* memory block might be configured in the real MPU by partial reconfiguration as the sequential configuration of multiple correctly aligned MPU regions, and this breaks the consistency of the original (unaligned) memory block. With these multiple subsets of the bigger memory block, the content is split over them. Problems arise if the content overlaps two subsets, for example instructions or memory accesses that result in unrecoverable memory faults because the partial reconfiguration will always enable one of the subsets and so always leave one part aside. In other terms, the partial MPU reconfiguration could trigger memory faults because of these overlapping instructions or memory accesses. Considering processor instructions, there could be three chained faults in a row because of this phenomenon: the instruction, first operand and second operand. We solve this issue by requisitioning additional MPU regions to cover all overlaps, so at most three additional regions. These MPU regions could be of the minimal size of an MPU region (32 bytes) or the next partially configured MPU region that would satisfy the MPU constraints and still stay within the bounds of the original memory block. For these reasons, the framework progressively requisitions the MPU regions with the highest numbers until passing the faulting instruction. At the next legitimate fault, the MPU is reconfigured like the user wanted it in the first place, and the execution continues. The process is completely transparent to the user, however implies a slowdown of the execution of some instructions to handle this case.

The second exception is related to the stack. ARM Cortex-M processors save the current context, that is a set of registers, in the current stack to later pop out the values again to restore the context. This happens during context switching, for example to handle an interruption or to switch active component. The issue is that the current stack is the stack of the currently executing unprivileged software component which is dependent on the active MPU configuration (if it were privileged, the background region would be enabled and there would be no issue). In the case that the current stack is not enabled in the MPU, it will trigger a memory fault when the context switching

routine tries to save the context on the disabled stack. However, the memory fault is also dependent on the current stack where the context is supposed to be stored once again, thus leading to a non-recoverable double fault: the context is lost. Thus, the stack must always be enabled in either ARMv6/v7 or ARMv8 processors. But there is a supplementary issue again because of the hardware constraints and the partial reconfiguration in ARMv6/v7. The situation is the following. If the block containing the stack becomes unaligned, because of a cut operation for example, the partial reconfiguration algorithm just enables in the MPU a part of the full unaligned stack. As the stack grows with the execution of the application, it will eventually reach the threshold of the partially configured MPU region. The partial reconfiguration algorithm then kicks in, however, cannot save the context on the stack, which leads to the double fault. Hence, the developer must ensure that the stack is always enabled in the MPU and correctly aligned at any moment of the execution. Obviously, the stack region is identified in the partial reconfiguration feature and is never kicked out of the MPU.

6.5 Conclusion

In this chapter, we presented technical solutions to implement the framework. A framework implementation requires four metadata structures: the *virtual* MPU structure, the sharing structure, the subblocks structure and the main structure. We demonstrated reasonable complexity for the system calls using these metadata structures. The services are not preemptible (atomic) by disabling interrupts during treatment. We leverage MPU features shared between different MPU versions and found in other MPU equivalent units in other architectures like the RISC-V PMP. We showed the framework can deal with additional hardware constraints on the MPU versions like the MPU region size or alignment constraints.

Chapter 7

Pip-MPU, adaptation of the Pip kernel via framework specialisation

Chapter outline

In this chapter, we present Pip-MPU, an adaptation of Pip for constrained devices, by specialising the framework presented in the previous chapter. Pip-MPU globally enforces a strict memory isolation security policy similar to Pip's (*cf.* presentation in Section 3.4.2 and below Section 7.2) while inheriting the flexibility offered by the framework. Pip-MPU will be originally presented in the article [56].

Fast reading track: Pip is presented in Section 7.2 and the connection is made with Pip-MPU particularly in Sections 7.3 and 7.4.3. Pip-MPU is then evaluated with results presented in Section 7.5.2.

7.1 Motivations

So far, we have focused on a flexible and secure memory management MPU module. However, memory management is not enough to run programs on conventional systems. Therefore, we are looking for a demonstration of the use of the framework with an adapted system that will benefit from the features of the MPU module. We choose to adapt Pip which current memory management is based on the MMU. Pip needs flexibility and has a very restrictive global security policy which makes it a good candidate to show the framework's possibilities. Pip is furthermore one of the lightest existing kernels as a protokernel, only featuring memory and control flow management, limiting the efforts of adaptation.

7.2 Pip

Pip is an OS kernel specialised in memory management and context switching. It is part of the protokernel family and a type of a separation kernel ensuring memory isolation between **partitions** (executable component).

Pip is sole privileged in the system while all the partitions run unprivileged in userland (see Figure 7.1). The partitions provide the missing features when necessary (for instance scheduling) as well as the applications and their support components. Pip joins high-assurance systems in that it is almost completely formally verified for memory isolation, that is there are proofs of the memory isolation developed in the Coq Proof Assistant [95, 40].

7.2.1 Partitioning model

Pip's memory management is based on a hierarchical partitioning model. A *partition* would typically reflect a process, a task or any executable piece of code along with its respective dependent data. A partition is thus composed by one or more chunks of memory. The main principle is that a partition can create one or several sub-partitions, that in turn can create sub-partitions. This creates a **partition tree**, as can be seen in Figure 7.1, which is a sort of genealogical tree of nested memory chunks. All partitions descend from the **root partition** which is the first partition to be created after the boot process. As an example, the root partition could be an operating system creating child processes. The other partitions are dynamically created during the system's lifetime.

7.2.2 Partition lineage

We define in this paragraph fundamental relationships within the partition tree, closely resembling a family tree. The relationship between a partition and a subpartition is referred as a **parent-child** relationship. A **parent partition** has **descendants** when a **child partition** also has children of its own. For the descendants, this parent partition is considered as an **ancestor**. The common ancestor to all partitions is the **root partition**. Partitions stemming from the same parent are **sibling partitions**.

7.2.3 Features

Pip exposes system calls that build up the partition tree:

- `createPartition`: creates a child partition
- `deletePartition`: suppresses a child partition and retrieves all memory pages it had
- `addVaddr`: shares a page with a child partition
- `removeVaddr`: removes a shared page from a child partition
- `prepare`: sets up a configuration page (holding kernel metadata structures)
- `collect`: retrieves a configuration page
- `countToMap`: returns the number of memory pages required to share (map) a memory page with a child partition
- `mappedInChild`: retrieves the child partition id with whom a page has been shared, if shared

- `yield`: switches context

A parent partition can create child partitions and share memory with them. All the memory attributed to a partition is directly accessible to this partition. The parent can only map memory that it owns. As a result, the whole system memory is divided into two parts: 1) Pip's memory and 2) the rest owned initially by the root partition. The root partition can access any partition's memory. This is a natural consequence of the hierarchical partitioning model, where the security of any partition is based on its ancestors, by recursion. As such, a parent partition has always the ability to tear down a child.

The parent decides which access permissions is associated with the shared memory. Indeed, a partition can restrict the use of the memory to a child partition. However, it can never increase permissions.

The partitioning tree is built up at runtime starting from the root partition.

7.2.4 Security properties

Pip's partitioning model ensures 3 security properties:

- **Vertical sharing**: a parent partition can share parts of its own memory to child partitions.
- **Horizontal isolation**: a memory page can only be shared to a unique child partition. This property ensures strict memory isolation between child partitions.
- **Kernel Isolation**: Pip's memory, as well as all metadata structures used for describing the partitioning model, is spatially isolated from the userland partitions. This property protects the kernel itself and ensures no partition can change its configuration by itself by bypassing Pip.

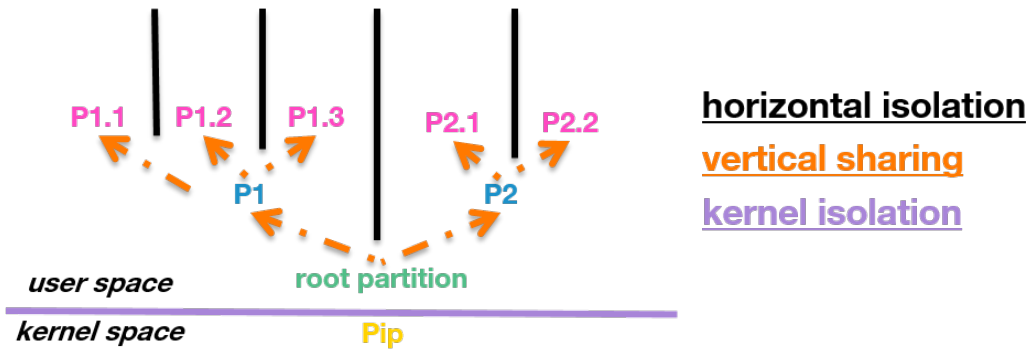


Figure 7.1: Pip's partitioning scheme.

7.2.5 Pip nomenclature

This paragraph aims at defining Pip kernel objects.

The metadata structures are Pip's configuration information, used to track the partitioning view and to configure the MMU with the correct information. Five internal structures are linked and used to operate any partition:

1. **Partition Descriptor**: A partition is identified by a *Partition Descriptor*, unique in the system for this partition. The Partition Descriptor is a metadata structure containing the partition's references to the other structures.

2. **MMU**: The MMU configuration is stored in the *MMU* structure. This structure is loaded in the MMU to control the partition's accesses. Additional information is given to each mapped page to know if it is present and/or accessible to the current partition.

3. **Shadow 1 (Sh1)**: The *Shadow 1* structure links mapped pages of the *MMU* structure to the child partitions. It mirrors (hence the name shadow) the *MMU* structure. The structure gives the information if the page is shared with a child partition and in such case which child.

4. **Shadow 2 (Sh2)**: symmetrically, the *Shadow 2* links each mapped page of the *MMU* structure to the parent's page memory that has been shared.

5. **LinkedList (LL)**: holds the configuration pages of the current partition in a linked list.

The kernel structures are reserved for the kernel, so never accessible from any partition (Kernel Isolation property).

7.2.6 Summary

In a nutshell, Pip is a security mechanism of type memory isolation (access control, confidentiality and data integrity) resting on a hierarchical TCB guarded by the MMU, separation of privilege and atomicity (hardware support to disable interruptions during service executions). It exhibits strong guarantees of isolation by the use of formal verification of the isolation property on its services using Coq.

7.3 Pip-MPU's requirements

As a kernel, Pip-MPU features more services and therefore more requirements than the framework. This section defines the requirements that Pip-MPU must satisfy. We classify

the requirements into four categories: security requirements, performance requirements, functional requirements and hardware requirements. Some requirements are directly inherited from Pip while others are only required to target resource-constrained low-end devices.

7.3.1 Pip's fundamental requirements

Pip-MPU inherits all Pip's requirements, except the ones tied to the MMU. Hence, we first state and classify the set of Pip's fundamental requirements.

- **SecReq1: Pip's security properties** Pip's security properties described in Section 7.2 shall be ensured.
- **SecReq2: Hardware-based memory protection** Any illegal access shall be blocked and identified by the hardware-based memory protection components. Only the kernel space has sufficient privileges to configure them.
- **SecReq3: Minimal software size** Pip's code must be minimal in size in order to be formally verified, to reduce the likelihood of vulnerabilities, and to ease the maintenance of the code base.
- **SecReq4: Limited access permissions updates** Pip shall ensure that only a parent partition can manage block access permissions (read, write, execution), that might be changed during the partition's lifetime. Pip shall ensure that a partition cannot increase the rights set up by the parent partition, on itself or one of its children.
- **FuncReq1: Flexible partitions** The partition tree shall be determined at runtime. Any partition can create and isolate a subspace of its own.
- **PerfReq1: Reasonable performance overhead** Pip shall maintain the performance requirements existing before the port to Pip in order to address real-world scenarios. This includes a fast startup sequence (fast cold start) that should not significantly impact the bootstrapping routine.

7.3.2 Specific Pip-MPU requirements

In a second step, we define additional performance and hardware requirements that stem from the constrained nature of the targeted devices.

- **HWReq1: MPU-based memory protection** Pip-MPU shall specifically use the MPU as hardware memory protection. As the MPU is only present in low-end devices, the corollary is that Pip-MPU only targets this class of devices.

- **HWReq2: No hardware modifications** Pip-MPU shall use hardware components present in Commercial Off-the-Shelf (COTS) systems, without any hardware modifications. This is to ease its adoption and reduce development time.
- **PerfReq2: Bounded execution time** Pip-MPU's algorithm complexities and implemented code shall be compatible with real-time constraints. Indeed, many low-end device scenarios have such constraints.
- **PerfReq3: Low memory consumption** Pip-MPU shall let enough space for real-world scenarios to fit in a Pip-MPU-based system. Pip-MPU's security overlay should be compatible with low-end devices' limited memory resources.
- **PerfReq4: Low power consumption** Pip-MPU's energy consumption overhead shall stay reasonable. Indeed, constrained devices are often powered on battery and the power consumption dictates their lifetime, as they are expected to operate in the wild for a long time.

7.4 Pip-MPU design

The adaptation is directed by the specialisation of the framework to Pip's security policy and makes the system Pip-like. It encompasses analogical equivalences and structural connections with the framework and implementation guidance.

7.4.1 Analogy between the framework and Pip-MPU

In the framework, userland components, executing in memory spaces, can create nested memory spaces out of their own memory space. In this way, the analogy is direct between Pip child partitions and nested memory spaces. Nested memory spaces in the framework are created on the fly, like child partitions, which satisfies FuncReq1. Furthermore, the framework is MPU-based without hardware modifications and as such perfectly fits COTS systems as required by SecReq2, HWReq1 and HWReq2. In addition to that, they both claim minimalism in line with SecReq3. For the framework, the compartmentalisation entity is specialised in providing only the minimal set of required memory isolation primitives that Pip-MPU can reuse to provide memory isolation respecting Pip's security requirements. At last, the computational complexities computed for the framework are in fact better in Pip-MPU because the services which had quadratic complexities (*cf.* Section 6.1) do not need anymore to span the whole partition tree to toggle the accessibility flag in all blocks requisitioned to hold protected metadata structures. With the Horizontal Isolation property, blocks to toggle can only be in direct ancestors, and not in parallel lineages because of a common ancestor that would

System call	Complexity
create/delete	$O(s * k)$
cut/merge	$O(s * k)$
add/remove	$O(s)$
prepare/collect	$O(s * k)$
activate	$O(1)$

Table 7.1: Complexities of Pip-MPU's memory management system calls. s is the number of structures in a memory space (expected <=8) and k is the number of direct ancestors (expected <=4).

have shared the same block between siblings. Hence, the new complexity Table 7.1 for Pip-MPU. But better than that, the number of direct ancestors reflects the number of isolation levels, which is expected to be at most 4. Because we previously analysed for the framework that the complexities of the services were bounded by linear complexities in the worst case, they are now bounded by a constant complexity. The new computed complexities fulfill the bounded execution requirements required by PerfReq2.

To summarize, the framework already satisfies the requirements FuncReq1, SecReq3, HWReq1, and PerfReq2. Furthermore, it partially satisfies SecReq1 because the nested memory spaces follow Pip's Vertical Sharing property and because the framework also protects the privileged compartmentalisation entity and its metadata structures responsible for the MPU configuration against userland accesses, which is equivalent to Pip's Kernel Isolation property.

The remaining security requirements (SecReq4 and Pip's Horizontal Isolation property) are covered by the framework's security policy specialisation in the next section. After that, the framework implementation is guided by all the remaining requirements related to performance metrics, which are evaluated in Section 7.5.

7.4.2 Framework security policy specialisation

The framework needs to be specialised to fully satisfy Pip's security requirements. The specialisation occurs within the system calls and in the metadata structures.

SecReq1 requires each child partition to be isolated from other child partitions stemming from the same ancestor partitions. We must operate a framework specialisation to restrict shared memory with and between child partitions to fully embrace Pip's security policy. We decided to reflect SecReq1 in the sharing structure. A unique field identifies the child partition with whom the block is shared. The system calls then retrieve this single value as the only possible child partition the current partition could share this block with. Hence, from the metadata structure itself, it is impossible to share a block with multiple children, satisfying the requirement. As a consequence

of the specialisation, we modify the framework's API. We removed the child partition identification for the system calls retrieving shared blocks since only one child can hold a shared block. SecReq4 requires restricting access permissions updates. In the framework, access permissions are set when adding blocks to a subdomain as done with Pip, but without restrictions. In our specialisation, the read, write and execute rights can never be elevated (but can still be lowered) guarded by additional logic in the block sharing system call.

7.4.3 Implementation guidance

For Pip's adaptation, we decided to keep the same nomenclature for similar objects, specifically the partitions for software components, and the main and sharing structure become respectively, the **Partition Descriptor** (PD) and the **Shadow1** (Sh1) structures. We also remove all MMU specific and optimisation structures from Pip: the MMU, the Shadow2 and the LinkedList structures. Only the MMU structure is (in an analogous way) replaced by the ***virtual* MPU** structure, also called **Blocks** structure. Following the same naming convention, the subblocks structure becomes the **Shadow Cut** (SC) structure. We also transfer the names of Pip's services to the framework's API, as shown in Table 7.2 . By taking over the same names for equivalent metadata structures and API, Pip-MPU revives Pip's conceptual frame. In addition to that, we add two more services: Pip's context switch service `yield` (adaptation performed by our team's engineer based on the work of another PhD student in our team [178]) and `findBlock` which scans the *virtual* MPU structure in search of a block containing a user specified address. While the first additional service is required to obtain all Pip's features, the second service gives the partition the possibility to inspect one of its memory blocks or the ones of its children. Another set of additional services is under development to offer the possibility to write in privileged registers while assuring isolation, they are not covered in this dissertation.

For formal verification purposes, and Pip particularly, we implemented the code directly in the Coq Proof Assistant. The framework's system calls are settled in pipcore. As every function had to be written in Coq for later verification, it had to be adjusted to a functional environment and recursive loops. This impacts performance as well, as recursive functions use more stack memory than loops.

For memory purposes, we decided to combine the metadata structures, instead of splitting them over several memory blocks as done in Pip where the structures are distributed over distinct memory pages. The rationale in Pip was to keep the MMU configuration separated from the rest of the blocks' attributes so to load the MMU directly by pointing to the new configuration. On the contrary, the MPU needs to be loaded register by register and there is a separate list of active memory blocks, so

Pip (MMU)		*Pip-MPU*	
Service	Target partition	Service	Target partition
`createPartition`	child	`createPartition`	child
`deletePartition`	child	`deletePartition`	child
`addVAddr`	child	`addMemoryBlock`	child
`removeVAddr`	child	`removeMemoryBlock`	child
`prepare`	child	`prepare`	child || *current*
`collect`	child	`collect`	child || *current*
`countToMap`	child	-	-
`mappedInChild`	child	-	-
-	-	`cutMemoryBlock`	*current*
-	-	`mergeMemoryBlocks`	*current*
-	-	`mapMPU`	child || *current*
-	-	`findBlock`	child || *current*

Table 7.2: Comparison of Pip (MMU)'s and Pip-MPU's memory management APIs. Equivalent services are on the same line. Note that Pip-MPU has services that target the current partition. The dashes represent the services that are not present relative to their counterpart API.

mixing the structures has no consequences and helps to reduce fragmentation. Hence, the *virtual* MPU, the Shadow 1 and the Shadow Cut structures are merged in a single **superstructure**.

For performance purposes, we enhanced the metadata structures to carry information decreasing the complexity of the system calls and overall speeding up the code. In order to significantly speed up the MPU configuration, we introduced a second MPU list, besides the list registering the manual MPU mapping through the `mapMPU` system call. This second list leverages the MPU packet configuration feature, allowing a fast configuration by setting up the MPU regions fourth by fourth. It consists of a pair of register values to be slammed directly in the MPU registers, instead of configuring each MPU entry one by one by retrieving the information in the metadata structures. This second list is always updated in the system calls, at the same time as the first list.

Furthermore, we limited the number of memory blocks in the metadata structures to 64 per partition. Rejection checks in the code implementation ensure this, setting the upper bound to the linear search in this structure. The linear search is needed in the `findBlock` service.

7.4.4 Implementation of the parent-child relationship

A partition is declared as a child as soon as a PD structure is set up for that partition. The partition inherits its identifier by the location address of the PD structure.

The number of memory blocks NB is limited by the number of superstructures NSS in a memory space times the number of entries NE of the *virtual* MPU (which is equal to the number of entries in Shadow 1 or Shadow Cut), so formally by the formula $NB < NE \times NSS$. Because the creation of a child consumes a block to store the new PD structure, the number of children NC a partition can have is then limited by the use of the memory blocks. More than that, a user would typically define a new partition with a minimum of 4 memory blocks (for the PD structure, a superstructure to register memory blocks and two memory blocks respectively for code and data). Theoretically, the number of children NC for a single parent is then limited even more finely by the formula $NC < NE \times NSS \div 4$. For our implementation, $NB = 8, NSS = 8$ then $NC = 16$.

Like in the framework, the PD structure holds a pointer to the parent identifier. In our implementation, we add the PDflag attribute to the Shadow 1 structure which indicates that a memory block is used as a PD structure. Since the Shadow 1 entries are linked to their corresponding *virtual* MPU structure entries, the start address in the *virtual* MPU entry gives the child's identifier without the need of explicitly writing it in the PDchild field (child in the framework). We give an example of a parent-child-grandchild relationship in Figure 7.2, to be compared with the presentation of the framework's metadata structures of Figure 6.2. Note that resemblance is high, naming conventions set apart. The implementation of the services sets up the global security policy of Pip using these metadata structures.

7.4.5 Summary

Pip-MPU inherits the flexibility and the compartmentalisation features of the framework and expresses equivalent requirements than Pip. We replaced Pip's memory management by the MPU module of the framework and adapted the structures and the API to stick to Pip's conventions. We introduced the specialised module within Pip's pipeline and made additional changes in the implementation to fully satisfy Pip-MPU's requirements which gather Pip's requirements and specific requirements due to the constrained environments. Thus, a Pip user can transparently use Pip-MPU for the majority of the services and will experience a similar system behaviour.

7.5 Evaluation of Pip-MPU

We evaluate Pip-MPU by implementing a prototype on a device based on an ARMv7 Cortex-M processor and by comparing it to a baseline scenario without Pip-MPU. The goal of the evaluation part is to answer the following questions: 1) Is the solution usable in practice to be implemented for constrained objects? 2) What are the solution's costs and benefits in terms of performance (processor cycles, energy consumption) and system

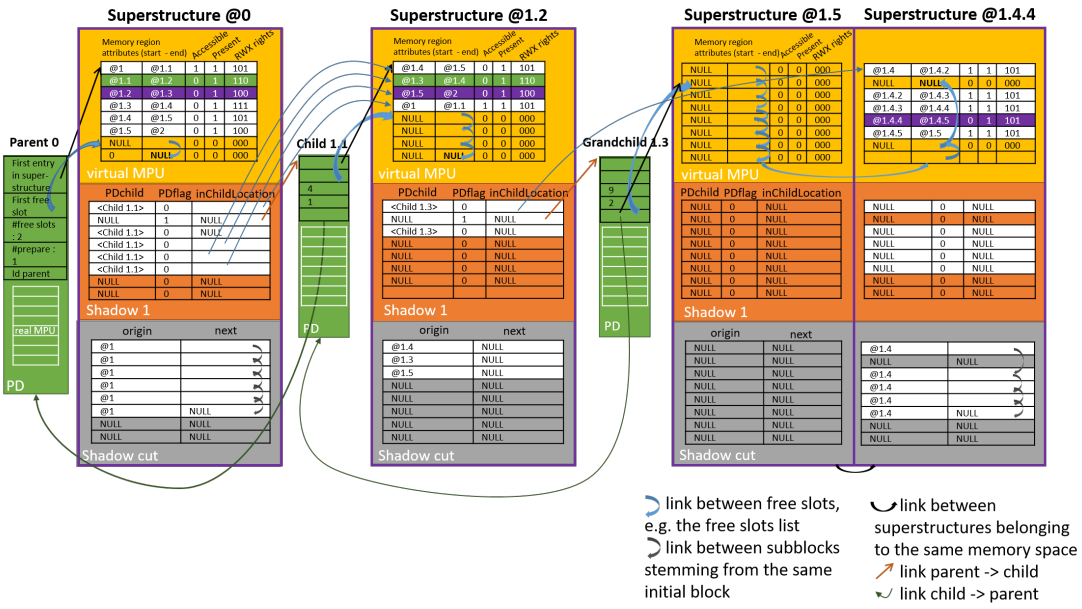


Figure 7.2: Example of a parent-child-grandchild relationship viewed from the perspective of the metadata structures. Partition 0 (parent) is parent to partition 1.3 (child of parent 0) which in turn is parent to partition 1.3 (grandchild of parent 0). Green blocks are used as PD structures in their respective child. Purple blocks are used as superstructures in their respective child or in their own partition (case of grandchild 1.3). The figure shows that block 1.4 has been cut several times (in blocks 1.4.2, 1.4.3, 1.4.4, and 1.4.5) which implies the original block 1.4 in the ancestors is inaccessible. For each memory block in all the partitions, we can observe there is no elevation of access permission rights compared to their original block, but can only be lowered (like block 1.3). Grandchild 1.3 shows a typical state after a `prepare`: a new superstructure is initialised out of block 1.4.4 from the same partition 1.3, it is placed at the beginning of the linked list of all superstructures related to the same partition, the first free slot is the first entry of the new superstructure and all entries have the default values except the last entry of the *virtual* MPU structure that connects with the free slots list with the previous one. Memory block 1 was shared memory between the parent 0 and the child 1.1. In our implementation, we consider a *virtual* MPU structure of length equal to eight (eight entries). Note the absence of order in the free slots list of grandchild 1.3 which resulted from previous `cut` and `merge` operations. Note as well the single reference between superstructures 1.5 and 1.4.4 compared to the multiple references in the frameworks between each inner structures.

overlay (size, lines of code, initialisation time)?

7.5.1 Experimental setup

Our prototype runs on an nRF52840-DK (Nordic Semiconductor) board [131]. The board is built around an ARM Cortex-M4 CPU (ARMv7-M architecture) running as fast as $64MHz$ with 1 MB of Flash and 256 kB of RAM, with an MPU composed of 8 MPU regions.

We perform static and dynamic analyses on 4 benchmark applications out of the Embench IoT benchmark suite applications [72]: aha-mont64, crc32, nsichneu, prime-count. We directly use the source files [63] without any modifications. They have been selected because the benchmark suite is free and open-source, the applications represent deeply embedded systems, they are compatible with our system constraints, they run bare-metal and they don't have any output streams. They also do not use the Floating Point Unit (FPU), even if one is present on our board, because our prototype did not support it at the time this evaluation was performed. The benchmark code is available online.[1]

The evaluation consists of two scenarios running an application 1) in Pip's root partition and 2) in a child partition. The root partition sets all applications in the unprivileged userland, making it impossible for them to run privileged operations. The child partition further restricts the memory attributed to the application, with the cost of abstraction. We compared each scenario against our baseline scenario consisting in running the benchmark application in the following configuration: privileged mode, without Pip and after the same system initialisation phase. The test application is regularly interrupted by the SysTick clock every 10 ms which triggers either a void handler in the baseline scenario or Pip-MPU's interrupt management handler in the Pip scenarios. As an end result, we present the total overhead induced by the use of Pip-MPU at different abstraction level for each evaluation metric. The CPU runs at a speed of $64MHz$ and each benchmark application is launched successively several times within a scenario to strengthen the results disparities and extend the experiment. An experiment associates a benchmark application with a scenario. We distinguish four phases in the experiment illustrated in Figure 7.3: the system initialisation phase (boot), the benchmark initialisation phase (the launch of the root partition and the child partition), the test phase that is the benchmark executing for several runs, and the benchmark end phase which stops the experiment and sends the collected data to the main computer driving the evaluation. Final post-mortem analysis is carried on with all the data collected from all the experiments to extract the information and generate

[1] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/benchmark/benchmarks

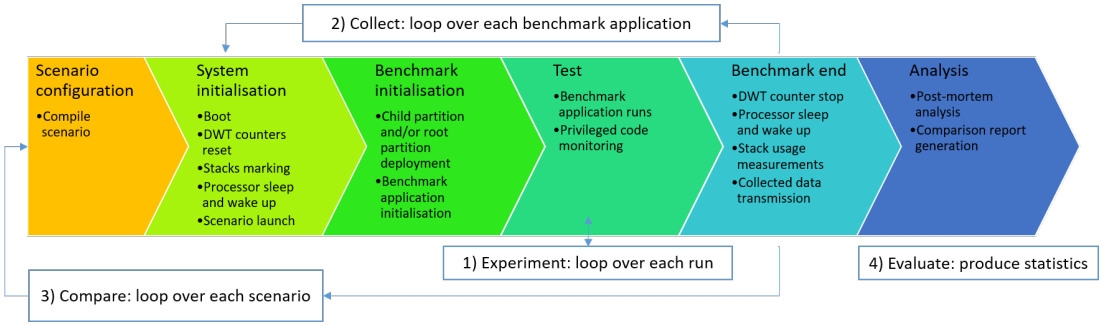


Figure 7.3: Evaluation phases. The evaluation consists in conducting the experiment on each benchmark application in each scenario and finally analysing all the data.

statistics reports.

7.5.2 Evaluation results

We wrote specific Python scripts to conduct the evaluation phase, cross-mixed and adapted from the scripts and tools offered by Embench and BenchIoT [2]. In this section, we describe the monitored metrics and how we collected the data. The final results present Pip's raw overhead in Table 7.3 and what performance costs to expect in Table 7.4.

The Source Lines of Code (SLOC) are the number of C lines of code counted after removing all comments and empty lines from the C source files by using the `gcc -fpreprocessed` option. They include lines containing only brackets, global variables and the function parameters that could spread on several lines (though remain limited). Table 7.3 presents the SLOC and size (in bytes) of Pip-MPU alone.

Stack usage is monitored by identifying the software components' stacks (main stack and app stack) and by marking them with a pre-defined value. As the stack is growing one address after the other, the last position where this value has been updated is the stack bottom address which witnesses the usage. In addition to the root partition's metadata structures, Pip-MPU's memory footprint also encompasses the metadata structures needed to create any runnable partition (the structures holding the list of blocks and attributes, as well as global partition data). The memory footprint is computed through formulas explained next. When the number of blocks in a partition grows by cutting or receiving memory blocks, the latter needs to be registered in supplementary structures of size S in bytes. Each supplementary structure can hold a constant number of blocks C, fixed at compile-time. Hence, for a partition of B blocks, we get $K + (B \bmod C) \times S$ *bytes* with K incompressible metadata. In our implementation, $K = 640$, $C = 8$, $S = 512$. As any partition requires a minimum of one metadata structure to hold the first blocks, it leads to a minimum memory footprint in RAM for each

	SLOC of C	Size (B)
Memory footprint in Flash		
pipcore (translated from Coq)	2483	5804
Pip handlers	789	908
MAL	843	1996
Pip init	71	772
Pip data + bss	-	64
Total Pip-MPU size	4186	9544
Memory footprint in RAM (B)		
Pip-MPU stack usage	516	
Metadata structures:		
- Per partition	$640 + (\text{B} \; mod \; 8) \times 512$	
- Min per partition	1152	
- Max per partition	4736	
Deployment (#cycles)		
	In root	**In child**
Pip-MPU initialisation	99022	165582

Table 7.3: Pip-MPU raw overhead. To compute Pip-MPU's size and memory footprint, the -0s optimisation flag was used.

partition of 1152 B, including the root partition. Furthermore, as the number of metadata structures for a partition is bounded by $MaxMS$ at compile-time, the maximum memory footprint for a given partition is $K + MaxMS \times S$. Applied to our system, it gives a maximum footprint of $640 + 8 \times 512 = 4736$ B. More than that, $MaxMS$ also dictates the maximum number of blocks a partition can hold with the formula $C \times MaxMS$. For our system implementation, a partition can register $8 \times 8 = 64$ blocks.

For the performance metrics of Table 7.4, we run the benchmark application configured for each scenario (baseline, in root partition and in child partition). Each time, we execute 3 runs in a row within the same experiment to collect data for at least 20 seconds (each benchmark application executes during 5-7 seconds). We launch each experiment 5 times and perform statistics on the results (average μ and standard deviation σ). The indicated overhead is the observed average overhead computed for each scenario compared to the baseline, *e.g.* the average on all benchmark applications of the average overhead on all runs.

The cycles count are retrieved from the Data Watchpoint and Trace (DWT) unit of the processor. We initialise the count just before the launch of the benchmark application and collect its value after the end of the initialisation phase and when the application is finished. The end of the initialisation phase marks the test phase, from where the benchmark application executes. For the baseline scenario, the initialisation phase

Metrics	In root	In child
Cycles		
Cycles overhead:		
i) in total	$\mu = 76302131$	$\mu = 74538344$
	$\sigma = 67494444$	$\sigma = 73634323$
	(+16.31%)	(+16.4%)
ii) during test	$\mu = 76203107$	$\mu = 74372762$
	$\sigma = 67495112$	$\sigma = 73634647$
	(+16.29%)	(+16.36%)
Privileged cycles over total cycles ratio:		
i) in total	$\mu = 0.86\%$	$\mu = 0.92\%$
	$\sigma = 3.8 \times 10^{-5}\%$	$\sigma = 3.3 \times 10^{-5}\%$
	(-99.14%)	(-99.08%)
ii) during test	$\mu = 0.87\%$	$\mu = 0.92\%$
	$\sigma = 3.9 \times 10^{-5}\%$	$\sigma = 3.3 \times 10^{-5}\%$
	(-99.13%)	(-99.08%)
Energy consumption during test		
Total energy overhead:	$\mu = 24.76\ mJ$	$\mu = 26.6\ mJ$
	$\sigma = 22.42\ mJ$	$\sigma = 23.00\ mJ$
	(+16.7%)	(+18.4%)
Energy overhead due to MPU:	$\mu = 0.05\ mJ$	$\mu = 0.07\ mJ$
	$\sigma = 0.16\ mJ$	$\sigma = 0.11\ mJ$
	(+0.03%)	(+0.04%)
Security		
Accessible application memory over total memory ratio:		
- Flash (code)	99.0%	6.27%
	(-1.0%)	(-93.73%)
- RAM (data)	99.35%	1.9%
	(-0.65%)	(-98.1%)

Table 7.4: Performances comparison (versus baseline). The test application is either executed in the root partition or in the child partition, compared to the baseline.

is almost void since it just calls the benchmark application. Moreover, the baseline scenario is always executing in privileged mode so the cycles count is fully privileged. On the contrary, in the Pip scenarios, the privileged cycles are monitored by counting the cycles only spent in Pip-MPU. We provide the ratio of privileged cycles over the total cycles from i) Pip-MPU's start and ii) only during the test phase. They are compared to the entirely privileged baseline.

The accessible memory areas represent the memory a partition has access to. The application in the privileged baseline has access to the whole memory whereas by using Pip-MPU the accessible memory areas are the blocks of the memory space. For the root partition, the accessible memory includes the whole memory minus the TCB (Pip-MPU and boot components). From there on, the root partition, as any other parent partition, decides which memory blocks to pass on to its children, thereby controlling their accessible memory areas.

The energy consumption is monitored using the Power Profiler Kit I (PPKI) [132] mounted on the nRF52840-DK board. Our testbed is illustrated in Figure 7.4. The PPK provides current measurements at $77kHz$ with 4 measures average that we multiply with a fixed voltage and integrate over time to get the total energy consumption. As the benchmarks use semihosting to send the performance data (cycles and stack usage) to the computer for analysis, the debugger remains active. However, no input or output is performed during the test phase. Furthermore, our set-up includes an additional nRF52840-DK board to interface with the PPK and send the measurements to the computer. We used a PPK library [185] to trigger the measurements because the desktop application was not stable enough for our experiments and it eased the integration with our python scripts. Nevertheless, as an upgraded version of the PPK (PPKII) was released some years ago, the library is not maintained anymore and the integration required to find a good match between the PPK's firmware version and the library and its dependencies. For our analysis, energy consumption is solely measured during the test phase. We mark this phase by setting the processor in deep sleep mode before and after the test phase and wake it up with an external timer. In this way, we can easily identify the test phase from the current measurements with significant current drops during the sleep phases (around 6mA during the test phase down to µA when sleeping), like demonstrated in the Figures 7.5, 7.6 and 7.7. The figures represent the variation of energy consumption during an experiment as a function of different scenarios using the same application. The part in red corresponds to the test phase. On these figures, we can clearly see the consumption drops before and after the test phase.

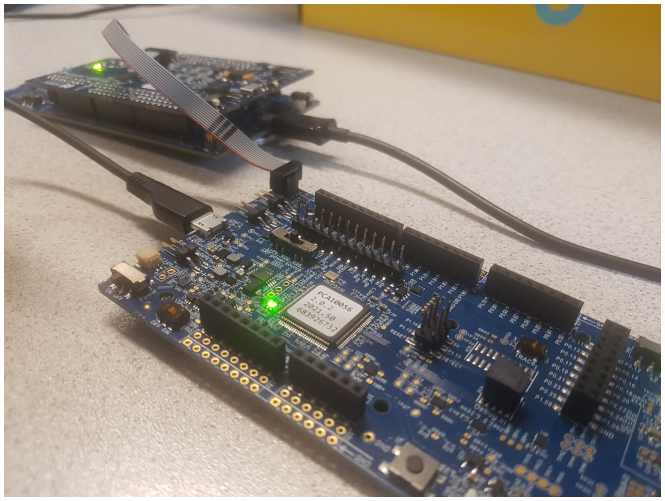


Figure 7.4: Our testbed. At the forefront, the nRF52840-DK board controlling the PPK. In the background, the nRF52840-DK board hosting the test application and on which is mounted the PPK.

7.6 Discussion and limitations

The figures presented in the previous section are valuable information to consider a port on Pip-MPU.

Pip-MPU takes respectively 1664 B (data, stack, root partition metadata structures) and 9544 B (code) of the available 256 kB RAM and 1 MB of Flash. It then fits easily the constraints of our targets (around 3.3% RAM and 3.8% Flash of Class 2 IETF devices) and leaves enough space for more complex applications, thereby fulfilling *PerfReq3*. Pip-MPU's minimalism, required by *SecReq3*, is therefore satisfied. Hence, we expect a good ratio for Pip-MPU's size relative to the size of rich OSes and their applications ported on Pip-MPU. To be noted, we considered scenarios with a correct test application, without triggering faults or using the partial MPU reconfiguration feature inherited from the nested compartmentalisation framework. We expect a stack usage increase in such cases.

The accessible memory areas metric shows the extent of the attack surface. In the baseline scenario, since the application is privileged, it can access 100% of the memory. On the contrary, when using Pip-MPU, the partition becomes unprivileged and is limited by the MPU. For the root partition, this value decreases by about 99%. Indeed, the root partition owns the whole memory except the parts reserved for Pip-MPU. The further away the active partition is from the root partition, the more the parent partition can restrict the accessible memory and the better the metric is. For the child partition in our implementation, we reduced its accessible memory area to respectively 2% and 6% of

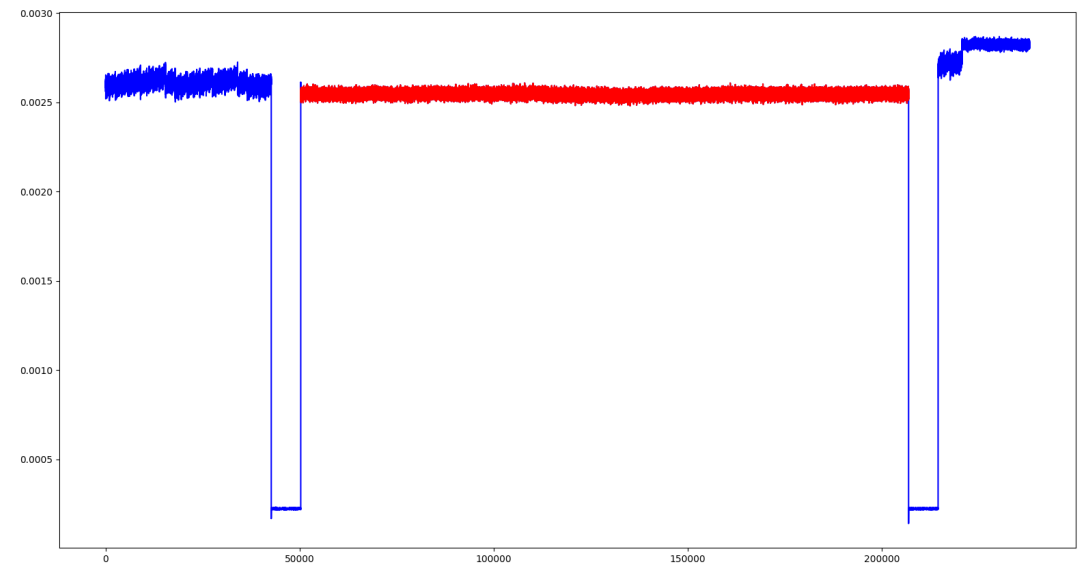


Figure 7.5: Baseline scenario. Energy consumption (mJ) as a function of time (samples).

the RAM and Flash areas. This means this child partition loses more than 94-98% of the memory that was accessible in the privileged baseline scenario.

The evaluation reveals a minimum memory footprint in a parent partition for each new child partition of around 1 kB for our implementation. This minimum should be increased because of the requisitioned entries in the parent partition to register the child's metadata structures. Indeed, the additional entries may not fit in the fixed-size metadata structure holding the block attributes, leading to the creation of a new metadata structure in the parent to host these entries (supplementary 512 B in our implementation). Other implementations can change the length of the structures (reducing or extending) by compilation options. Because some blocks become inaccessible when hosting metadata structures, one could pick a number of entries slightly bigger than the number of available MPU regions since the MPU can only be configured with accessible blocks. This is the case in our implementation because we choose a length equal to the number of available MPU regions (eight), however the customised memory fault handler needs some reserved MPU regions to deal with ARMv7-M MPU region alignment and size requirements. This is valid for any implementation, because the number of available MPU regions is implementation-defined and because some architectures like ARMv8-M do not have the same strict hardware requirements.

Pip-MPU's raw overhead is declined in two stages: the initialisation phase (for the root and child partitions) and the test phase (the running application). The initialisation phase shows an averaged initialisation phase lasting 99022 cycles ($1.5ms@64MHz$) and

165582 (2.6ms@64MHz) respectively, for the root partition and the child partition. This represents the pure overhead of Pip-MPU's initialisation time over the baseline, resonating with *PerfReq1*. Furthermore, we observed an execution overhead for the test phase of about 16 % caused by Pip-MPU's restoration context sequence when receiving the SysTick interrupt. This latter value should be appreciated within the tested scenario and values are expected to be higher for a rich OS ported on Pip because of multiple interrupts causes. While the performances proved sufficient in the evaluation, there are potential improvements areas to further optimise the system calls if deemed necessary in the future by adding optimisation metadata structures similar to Pip. In addition to that, Pip forces the benchmark application to run in unprivileged mode. We observe a drop of more than 99% of the privileged cycles when using Pip that corresponds to Pip's execution. The opportunities to exploit the privileged operation mode are reduced as much.

Energy consumption resulted in a 17-18% increase when using Pip-MPU. Moreover, we launched the benchmarks while switching off the MPU. It showed a consumption decrease of 0.02-0.2% depending on the scenarios. It demonstrates that the MPU use (due to the context switching and permanent protection) does not impact significantly the power consumption, contrary to what has been argued to explain the non-use of the MPU [194]. These measurements are important for IoT devices that may operate in areas without power line access and thus depend on a limited power battery. They satisfy the final requirement *PerfReq4*. About the energy consumption, the PPK is sampling at 77kHz, so almost 3 orders of magnitude lower than the actual processor frequency clocking at 64MHz. This makes it impossible to detect all energy consumption peaks. Indeed, at that frequency, a peak that lasts no more than 13µs stay undetected. However, the peaks must be a frequent phenomenon to disturb the statistical mean of our measures because our experiments last 20-30 seconds. Therefore, we argue here, in an informal way, that our results are valid.

Other metrics are proposed in BenchIoT but are not evaluated here for the following reasons. First, we did not evaluate the number of sleep cycles as Pip never puts the CPU into sleep. Second, we did not include Data Execution Prevention (DEP) or the enforcement of the $W \hat{} X$ security principle, because Pip does not set them up. Indeed, the existence of such or additional security principles (like deciding which memory blocks to isolate) are strict partition design choices. Third, ROP gadgets and indirect calls are known techniques for an attacker to take control of the control flow and perform impactful attacks [153]. We evaluated the ROP gadgets and indirect calls overhead, respectively to 1780 and 9 due to Pip-MPU (directly using BenchIoT's tools based on [103]). However, we do not recognise them as relevant for Pip-MPU. The rationale is that Pip-MPU's or ancestor partitions' code and data are private and invisible from the

point of view of the active partition. Illegal access trials by crafted ROP gadgets end up in MPU memory faults caught by the ancestors. Furthermore, pipcore being developed in Coq before C translation, it holds characteristics of a functional programming language like a higher number of functions degrading these particular metrics. Hence, they do not represent for us relevant metrics. It should be noted though, that Pip-MPU does not prevent ROP attacks within the partition but against Pip-MPU and the partition's ancestors. Fourth, we did not single out privileged cycles and SVC cycles as they represent the same thing for Pip-MPU. Indeed, Pip-MPU's entry points are the SVC and are the only privileged code that can run after the initialisation phase.

As a result, the preliminary analysis and the evaluation showed full compliance to Pip's requirements and requirements expected for resource-constrained devices. Impactful security measures like privilege segregation of user and kernel/sensitive code are sometimes not used to lower production costs or reduce energy consumption. We showed simple applications such as those used in our evaluation can directly benefit from Pip-MPU's protection with almost no effort.

The scenarios explored in the benchmarks have a maximum of one isolation level. This is sufficient for bare-metal applications but we expect another level when porting an OS. A supplementary level implies additional abstraction to go through the partition tree that might degrade performance.

Pip-MPU entails the presence of an MPU which is a strong limitation for embedded systems without MPU. However, previous works [33] showed the MPU is present most of the time in Cortex-M3/4/7-based micro-controllers, thereby supporting the applicability of Pip-MPU. In addition to that, the compartmentalisation framework is generic to systems supporting privileged mode segregation and has an equivalent unit to the MPU. We believe our approach is then reproducible on processors from other vendors providing equivalent features.

7.7 Defence against attackers and assumptions

Attackers identified in our attacker model 1.2 are not able to compromise Pip-MPU. Indeed, when the system boots, it first initialises the hardware, including the MPU, and Pip-MPU follows. Pip-MPU leaves kernel space and kicks off the root partition in user space (the same procedure as in Pip). At that point in execution, the system is in a safe state as initialised by Pip-MPU, with enforced access permissions to the unprivileged root partition. All future operations are MPU guarded by Pip-MPU. Attackers now have the lead but are limited by the MPU to perform their attacks. The compartmentalisation set up by Pip-MPU in child partitions prevents the attackers from further harming the system outside the compromised component.

For the previous statements to hold, we must make some assumptions. First, we must trust the firmware loader (*e.g.* flashing of the code on the target device). Then, we must assume a correct bootstrapping routine, *e.g.* the start procedure is considered reliable and always leaves the system in the intended initial state. We also need to trust the compilation toolchain, IDE and used tools, and the hardware platform (MPU included) which should work as described in its specification. We only target 32-bit single-core ARM-based microcontrollers provided with an MPU enabled. Furthermore, as local permissions are changed during runtime, the system developers, or attackers, must have access to the source code of the sole component they are responsible for or the full source code. These assumptions are realistic and the basis for many secure systems. Further discussions on assumptions are discussed in the next chapter, enriching the list to include the formal verification.

7.7.1 Conclusion

Pip-MPU is the Pip kernel variant based on the Memory Protection Unit (MPU) instead of the Memory Management Unit (MMU). Pip-MPU does not require any hardware modification on Commercial Off-the-Shelf (COTS) systems and only depends on the MPU, which implies privilege separation (privileged/unprivileged modes). The memory isolation offered by the MMU is transferred into MPU-based memory isolation by specialising the framework to inherit flexibility with the MPU and to satisfy Pip's security requirements.

We presented an implementation guidance that is also portable to other ARM architectures such as the ARMv8 Cortex-M architecture.

Our evaluation, after full implementation on an ARMv7 Cortex-M device, shows that Pip-MPU reduces the attack surface from 100% down to 2% while requiring 10 KB of Flash, 550B of RAM and an overhead of 16% on both performance and energy consumption.

The framework showed compatibility with a system requiring strong flexibility and security.

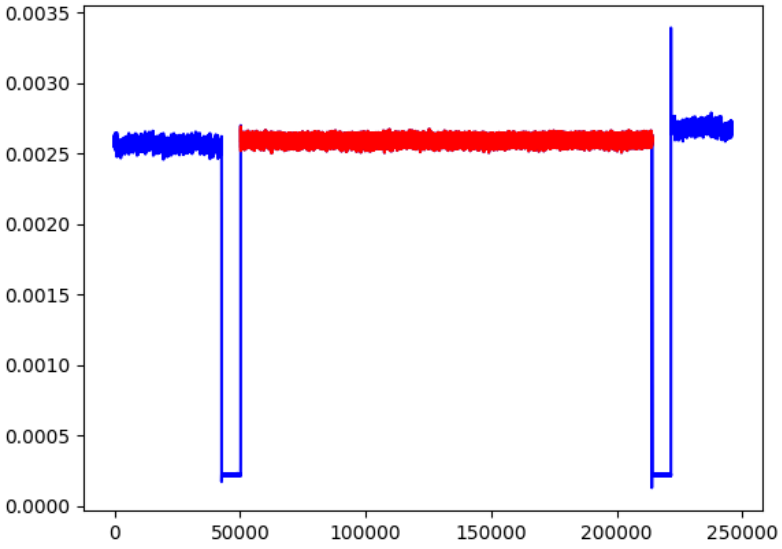


Figure 7.6: In root partition Pip scenario. Energy consumption (mJ) as a function of time (samples).

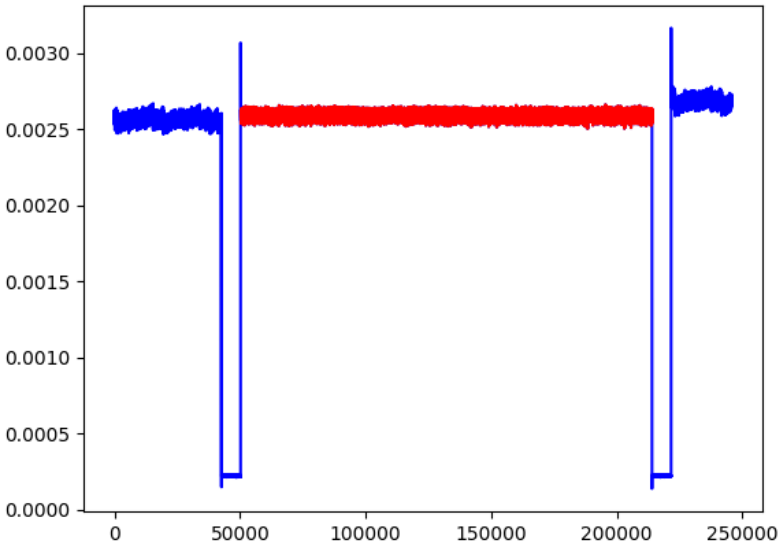


Figure 7.7: In child partition Pip scenario. Energy consumption (mJ) as a function of time (samples).

Review of Part II

Chapter outline

Review

Pip-MPU is Pip (MMU)'s variant adapted to constrained devices featuring a Memory Protection Unit (MPU). Pip (MMU) is one of the simplest existing kernels offering strict memory isolation. With Pip-MPU, it is Pip's security-by-design that is transferred to resource-constrained devices.

Pip-MPU extends the list of secure MPU-based memory isolation solutions. However, it offers much more flexibility and is more portable. These notions are of utmost importance for the expanding heterogeneous complex IoT ecosystem. IoT devices are expected to support virtualisation [77], multi-tenancy [24], might change ownership during their lifetime, might evolve [92, 67] and are subject to attacks or serve as a medium for massively distributed attacks [112]. The devices would benefit from locally and dynamically established permissions. As each nested memory space is based on the least privilege principle, it is creating as many implicit privilege levels. For IoT security, this means any interface with the outside world can be contained in a partition guaranteed by the memory protection of the MPU. Pip-MPU is the first OS for constrained devices to demonstrate such flexibility while ensuring a very strict memory isolation policy.

According to ARM, for Cortex-M processors, "there is always a trade-off between performance, features, against silicon size and power" [12]. Pip-MPU targets this family of processors and does not give up security while exhibiting reasonable performance downgrades.

Pip-MPU is a specialisation of a versatile framework we designed. Agnostic and portable MPU solutions are ongoing works in the kernel community [176]. We demonstrated compatibility of the framework with ARMv7 Cortex-M-empowered devices in Pip-MPU's implementation, but claim its portability on ARMv6-M since the MPU programmer model is close, but also ARMv8 Cortex-M-based devices and RISC-V PMP because the leveraged features are widespread among these equivalent units for constrained devices. Because ARM dominates this slice of the IoT market, Pip-MPU is available for most of the deployed devices. The framework could be used to resketch existing solutions in order to inherit the offered flexibility by specialising their respective security policies in the implementation of the services.

Pip-MPU is expected to host applications in need of strict memory isolation and high flexibility. We acknowledge potential difficulties in porting existing systems upon Pip-MPU. The reason is that OSes for constrained devices and their applications usually run in the same address space and at the same privilege level (which makes them vulnerable to total compromission by an attacker) [105, 34]. They are by design not suitable for executing in the unprivileged mode and in isolated address spaces. However, our team successfully ported the RIOT-OS [22] on Pip-MPU, with ongoing works on porting additional drivers, which constitutes a proof of Pip-MPU's usability but also the basis of reproduction for future ports. The port gives us feedback to optimise performances, with already identified leads in modifying Pip-MPU's metadata structures, if deemed necessary. Pip's flexibility can also be leveraged to create a secure-by-design architecture for containers on low-end devices [23]. This use case differs from the typical use case for low-end devices, which consists in isolating multiple code components within a single-thread and multi-tasking bare-metal application because it involves multiple parties and requires reconfiguring the memory partition during the device's lifetime.

Finally, low-end IoT devices are broadly used for embedded systems and the trend shows an increase of IoT devices in the next years [115]. Pip-MPU aims to participate in safer intelligent environments by hosting secure applications for low-end devices. As vulnerabilities will certainly continue to strike the ecosystem in the near future, which is the lecture of governments and the industry that actively invest in cyber security, our proposition is in line with an immediate hardening of deployed constrained devices.

Perspectives

With Pip-MPU, we now have a system at least as secure as state-of-the-art solutions.

Pip-MPU claims compatibility with other MPU versions and processor architecture, which could be demonstrated with a port on ARMv8 Cortex-M empowered devices or on RISC-V and leverage the Physical Memory Protection (PMP) unit.

However, one can also ask how really secure the system is. We have already spanned over our assumptions, but there are still open questions on security. Indeed, even with a minimal TCB, how can we be sure there are no flaws in the design? How can we know for sure Pip's security properties are well implemented in Pip-MPU? Testing Pip-MPU showed only expected results, but did we cover all the cases? What are the security guarantees of the system? All in all, we *might* have a secure system, but we need to trust many elements, including the design itself. There is a way to transform the trust in the system into a mathematical certitude. In the next chapter, we formally verify Pip-MPU against Pip's security properties.

Takeaways

- Low-end IoT devices are vulnerable to impactful attacks
- Current hardware-based security solutions do not offer the flexibility required for new use cases in the dynamic IoT ecosystem
- Pip-MPU is the protokernel Pip adapted to constrained devices
- Pip-MPU is the specialisation of a framework we designed that sets up flexible nested compartmentalisation customised with Pip's security policy
- To our knowledge, Pip-MPU is the first and smallest isolation kernel for resource-constrained devices which provides flexible nested compartmentalisation

Part III

High-assurance/trustworthiness: formal proof of a kernel's memory isolation on constrained devices

Introduction to Part III

In this part, we formally verify Pip's security properties on Pip-MPU. The proofs elevate our confidence in Pip-MPU's strict memory isolation scheme by acquiring a mathematical certitude.

There are different ways to see a system as secure. Apart from the definition of security (*cf.* Section 2.2), one can test, more or less extensively, the security claim with debugging tools or testing tools. However, these methods do not cover all the possible cases and the corner cases might introduce vulnerabilities as described in the Preliminaries 2.

Formal methods do cover all cases. Formal verification enhances a system with mathematical confidence about some specified properties. The formal claim is independently machine-checked. Anyone can challenge the proofs, and since the verification is done by a reliable software, can have a high confidence in it (*cf.* Section 2.2.4).

Pip-MPU follows Pip's proof workflow and formal proofs are conducted in the Coq Proof Assistant [95]. Pip's security properties constitute proof invariants (*cf.* Section 2.2.4) that need to be verified for each Pip-MPU service.

It is usual for spatial separation system to be verified through a proof assistant because the aim is a full verification for security-critical environments, they have small software TCBs suitable for full formal verification, it is hard to represent high-level complex properties with model checking languages, and because the last twenty years exhibit successful projects using that method [192].

In the following, we first describe Pip-MPU's hardware model and related assumptions. We express Pip's security properties and structural properties with Pip-MPU's model. We demonstrate the formal proofs of the security properties on several Pip-MPU services. The properties are first informally demonstrated before presenting machine-checked formal proofs in the Coq proof assistant. We finally develop metrics to follow the progress of the proof process and combine them to obtain the best-effort path to achieve a complete verification.

Chapter 8

Context

Chapter outline

Fast reading path: the reader already acquainted with the Coq proof assistant and Hoare logic can skip the Background section. I recommend however, to take a look at the explanation of Pip's proof workflow in Section 8.3.

8.1 Motivations

A system running on a Pip-MPU-empowered platform must have confidence in Pip-MPU. But why would a user trust Pip-MPU? Some answers have already been developed in the previous part: Pip-MPU is hardware-rooted in the MPU, is open-source, satisfy the requirements on Pip's security properties and we conducted a security assessment.

While it does give a sense of confidence in the design and the security, it might not be enough for high-assurance systems. Sure enough, open-source code and a large community is enough to create trust but not enough to avoid vulnerabilities, as can be experienced with Linux [43]. Furthermore, informal requirements checks might have missed a corner case introducing a vulnerability. Moreover, unit testing the implementation might not have covered all the cases, especially at a higher level of abstraction like compartmentalisation.

Using formal methods remedy this situation by giving an independently machine-checked proof of the security claim in a very reliable way. In particular, security in Pip-MPU relies on the correct use of the MPU replicating the partitioning tree's properties. A hardware-based formally verified computing base gives the highest confidence to the system.

A system based on Pip-MPU has a security that is hardware-rooted in the MPU and is software-rooted in formally verified Pip-MPU.

8.2 Background

8.2.1 Hoare triple

Pip (MMU) uses Hoare logic [90] to formally verify its services. Hoare logic gives a formal system to conduct the proofs of properties of a program, *i.e.* rigorous deductive reasoning rules of inference.

In Hoare logic, the formal specification refers to a **Hoare triple** composed of a program Q, a pre-condition P and a post-condition R. The pre- and post-conditions are general assertions (properties) on the values of the program written in mathematical logic.

The Hoare triple is formally expressed as $\{P\}Q\{R\}$.

It logically expresses that if the program Q executes and terminates, then its post-conditions R are true at completion if the pre-condition P was true before Q executed.

Hoare triples follow the rules of inference involving consequence (for $\{P\}Q\{R\}$: if the post-condition assertion R logically implies assertion S, then $\{P\}Q\{S\}$, and inversely, if the pre-condition P is implied by an assertion O, then $\{O\}Q\{R\}$), composition (break down a program sequence by sequence and connect Hoare triples on each sequence to prove the full program, *i.e.* for a program of sequence $(Q1\ ;\ Q2)$ connecting the post-condition of the first sequence's Hoare triple with the pre-condition of the second sequence's Hoare triple: $\{P\}Q1\{R1\}$ and $\{R1\}Q2\{R\}$ then $\{P\}(Q1;Q2)\{R\}$), iteration (the controlling condition of a loop is false when leaving the loop).

Other researchers extended Hoare logic and proposed new formal elements. Among them, Dijkstra proposed the *weakest pre-condition* (respectively strongest post-condition), which is the least restrictive (resp. any post-condition) statement that satisfies the post-condition (resp. is implied by the pre-condition) of a Hoare triple.

Pre- and post-conditions can hold the same properties: these are the invariants (*cf.* Section 2.2.4 on formal verification).

8.2.2 Proofs in the Coq Proof Assistant

Proof assistants are tools that help the developer to write and conduct proofs.

Coq [95] provides an interface to write programs and proof script in the Gallina functional programming language. A proof script that satisfies the **proof goal** provides a **proof term**.

Coq includes a small kernel that validates the constructed proof term against the proof goal, *i.e.* type checking and statement matching.

Proof development in Gallina is not easy and so is simplified by a high-level proof script language called *Ltac* which terms are called **tactics**. During the proof development, the developer can introduce **axioms** to describe the formal model or facilitate the proofs, and even leave aside some proof obligations by the **admit** tactic. However, a proof script starting by the keyword $Proof$ and ending with *Qed* cannot include any *admit*ted terms. In such a case, the only way to end the proof (for now) is with the keyword $Admitted$.

In Coq, one can write theorems and lemmas to solve specific proof goals exhibiting the same pattern.

Hence, Coq helps the developer to both write programs and proof scripts. Coq enables to construct valid proof terms in Gallina and tactics that satisfy a proof goal. The proof goal can be further divided into sub-goals, for example by the use of Hoare logic with intermediate statements to prove.

8.3 Pip's workflow

Figure 8.1 shows the two paths in Pip's workflow: proofs and executable.

So far, we have not discussed about the hardware model since we executed the services on the hardware directly. However, proofs must be conducted on the side effects of the services. In other words, the proofs must know the services' effects on the hardware to verify the properties. The proofs need an abstract hardware model to work on. On the contrary, the executable runs on the real hardware.

Note that in Pip, the proof workflow does not start from the C implementation but from the model up to the proofs, whereas the executable workflow starts from the model down to the executable.

Executable workflow The real hardware is the target for the executable workflow. First, the code of the services is *automatically* translated into the C language, literally word by word, by a custom tool named Digger. Then, the low level primitives of the HAL are *manually* written in C equivalent code. For obvious reasons, the hardware model is not translated in C as we are using the real hardware. From there on, the C

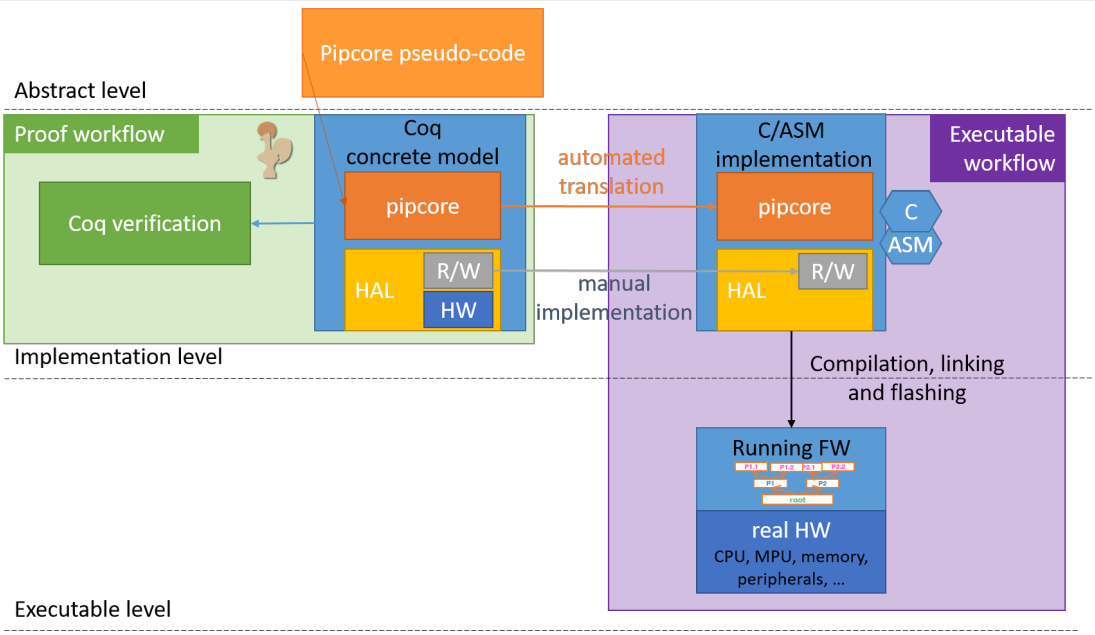


Figure 8.1: Pip workflow: proofs and executable. The proof workflow directly uses the concrete model, composed of the services and a hardware abstraction layer (HAL), to conduct the proofs at implementation level. The executable workflow first needs to translate the services and the low level read and write primitives (R/W) in C, before cross-compiling to the target platform. To be noted, the hardware model of the concrete model (HW) is not translated during the executable workflow, as the target is the real hardware and not a simulator.

implementation is compiled, linked to get an executable. The latter is finally flashed on the target platform.

The executable workflow has been used in the previous part to generate the executable of Pip-MPU.

Proof workflow Typically, the proof workflow kicks in after the design. Then comes the specification of the properties to prove and finally their formal verification. In Pip (MMU), but not in Pip-MPU as will be explained later, there is a code and proof co-design approach.

Proofs are always conducted on an abstract view of the reality. However, the abstract level depends on the proof's methodology. Pip's proof methodology is to conduct proofs at implementation level, directly on the code of the services. Thus, the abstract level is nevertheless a concrete model.

For that, the abstract model is composed of services and a Hardware Abstraction Layer (HAL). The services use the HAL as a library to interact with the hardware, among which the MPU, with elementary hardware access primitives (reads and writes). In our model, the HAL interacts with an abstract view of the hardware (and so proofs are conducted on this abstract view) but in the implementation, the HAL interacts with the real hardware.

All in all, we have the following model:

- the services update the kernel structures by using the low level primitives of the HAL
- the HAL interacts with the hardware model to change the abstract system state
- the system state holds properties (including the security properties) verified in the proofs

Chapter 9

Intuition and formalisation

Chapter outline

This chapter focuses on the proof workflow and the connexion with the executable workflow.

First, we expose the proof goals, which are the verification of Pip's security properties for each Pip-MPU service. The security properties are expressed assuming correct Pip metadata structures and inter-partition relationships. We thus include invariants that guard the correct structure of all partitions. These additional invariants and Pip's security properties form the isolation invariant.

Second, we discuss the followed proof methodology to prove the isolation invariant.

Third, we give to the reader the intuition on the proof of the security properties by exploring the sketch of proof of two services, one which does not modify the state and the other that modifies it.

Four, we present the formalisation in Coq of our proof model.

Five, we demonstrate the formation and propagation of properties, which constitutes the formal basis for the proof of the security properties (see next Chapter 10 on the formal proof of the security properties). The demonstration uses our Coq formal model.

Finally, we connect our model to the C implementation and the real hardware used by the executable workflow.

Fast reading path: this chapter introduces many formalisation elements. I recommend at least taking a look at the sketch of proof of the security properties for two representative Pip-MPU services from Section 9.3. I focus on their proofs in Coq in the next Chapter 10. For details on proofs in Coq at the primitive levels, that propagate low-level properties, read also Section 9.5. Finally, it might be interesting to see how the Coq model connects to the C code used to compile the final executable, which is explained in Section 9.6.

9.1 Proof goals

Pip-MPU has a different design than Pip (MMU) but still aims to prove Pip's security properties. The security properties are invariants, *i.e.* properties that must be verified whatever the system state. The security properties extract information from the metadata structures to construct the partitioning model on which they can reason about. Consequently, they can only be proven on a sane/correct partitioning model which implies the metadata structures are correctly formed. The consistency properties ensure

the latter. They check the correctness of the configuration of the metadata structures. The consistency properties are also invariants.

Pip-MPU has then two groups of proof invariants: the security properties and the consistency properties. We formally define them in this section.

Furthermore, the proofs do not only serve the abstract verification of the correct isolation in the partition tree. Indeed, the security of the system is established by the correct configuration of the MPU, that effectively sets up the security properties to the real system. The link between the two, MPU and correct configuration, is made in the formal verification. Hence, the proofs aim to verify that the memory blocks loaded in the MPU adhere to the global partition tree model and that this partition tree itself satisfies Pip's security properties.

9.1.1 Formal expression of the security properties S: HI, VS, KI

Pip's security properties are: Horizontal Isolation (HI, also called partitionsIsolation), Vertical Sharing (VS), and Kernel Isolation (KI) (*cf.* Section 7.2.4).

Special case of KI

The full Kernel Isolation KI security property is in fact split in two: 1) the kernel code and data isolation from userland partitions and 2) the protection of the metadata structures (here named Kernel Data Isolation by inheritance). Only the Kernel Data Isolation property must be proven because we describe informally here that the partitions never access kernel code and data. Indeed, partition memory is only given by ancestor partitions originating from the root partition. Hence, partitions only own strictly less memory than their ancestors, and so does the root partition. When Pip-MPU starts, the root partition is loaded with the system's memory space *outside* kernel code and data memory regions. Thus, the only property that remains to be proven for Kernel Isolation is Kernel Data Isolation (hereafter named KI instead of full Kernel Isolation).

Formal expressions

To express the properties, we formally define some design elements. We refer to the global partition tree $PartTree$. The memory blocks (accessible or not) owned by a partition $part$ form the set $MappedBlocks[part]$ stored in the *Blocks* structure. The subset of memory blocks that are accessible are called *AccessibleMappedBlocks[part]*. The blocks holding metadata structures (configuration blocks, not accessible) are referred to as $ConfigBlocks[part]$. Recall that *cut* blocks are not accessible in the ancestors because we anticipate that they will become configuration blocks, however, do not

play any role in the security properties. The children are referred to as $children[part]$. $AllPaddr[blocks]$ lists all addresses contained in the $blocks$ list.

Definition 9.1.1 (Vertical Sharing VS)**.**

$\forall parent \in PartTree,$
$\forall child \in children[parent],$
$\forall blockAddrChild \in AllPaddr[Blocks[child]],$
$blockAddrChild \in AllPaddr[Blocks[parent]].$

Definition 9.1.2 (Horizontal Isolation HI)**.**

$\forall parent \in PartTree,$
$\forall child1,\ child2 \in children[parent],$
$\forall blockAddrC1 \in AllPaddr[MappedBlocks[child1]],$
$blockAddrC1 \notin AllPaddr[MappedBlocks[child2]].$

Definition 9.1.3 (Kernel Data Isolation KI)**.**

$\forall partition1,\ partition2 \in PartTree,$
$\forall blockAddr \in AllPaddr[AccessibleMappedBlocks[partition1]],$
$blockAddr \notin AllPaddr[ConfigBlocks[partition2]].$

Note that KI is not defined as a parent-child relationship because the isolation of metadata structures from accessible memory blocks must also be satisfied within the partitions that hold them. To be noted as well, proofs are conducted considering the whole set of blocks and not only the active blocks configured in the MPU, because all blocks are eventually eligible to be mapped in the MPU.

9.1.2 Formal expression of the consistency properties C

We classify the consistency properties in three categories: 1) inner structural type 2) inner structural set 3) inter-partition. We developed a set of 27 consistency properties, all depending on the memory state s. We formally define each property in the following.

For our formalisation, we write $structure.field$ for the value of field $field$ in the metadata structure $structure$. Fields pick up the field names of the metadata structures

described in Section 7.4 and use the Blocks structure denomination to refer to the *virtual* MPU structure. The memory model m is composed of elements *element* of type T that can only be kernel elements of type PD (PD structure), BE (memory block entry), $Sh1E$ (Shadow 1 entry), SCE (Shadow Cut entry) or physical addresses of type $PADDR$. We write $m.@$ for the value located at address entry @ in m and $TypeOf(m.@) = T$ to indicate the type at that address. We refer to a default *element* as $DEFAULT(element)$ if their inner elements contain only default values.

Inner structural type properties

- nullAddrExists s
- wellFormedFstShadowIfBlockEntry s
- wellFormedShadowCutIfBlockEntry s
- PDTIfPDFlag s
- AccessibleNoPDFlag s
- FirstFreeSlotPointerIsBEAndFreeSlot s
- BlocksRangeFromKernelStartIsBE s
- KernelStructureStartFromBlockEntryAddrIsKS s
- sh1InChildLocationIsBE s
- StructurePointerIsKS s
- NextKSIsKS s
- NextKSOffsetIsPADDR s

Definition 9.1.4 (nullAddrExists)**.** Address 0 is a special entry in the memory model taking on the role of the empty address. It must be of type $PADDR$.

$$TypeOf(m.0) = PADDR$$

Definition 9.1.5 (wellFormedFstShadowIfBlockEntry)**.** $Sh1E$ is linked to a particular BE by an offset $Sh1offset$ in the metadata superstructure.

$$\forall @, TypeOf(m.@) = BE \implies TypeOf(m.(@ + Sh1offset)) = Sh1E$$

Definition 9.1.6 (wellFormedShadowCutIfBlockEntry)**.** SCE is linked to a particular

BE by an offset $SCOffset$ in the metadata superstructure.

$$\forall @, TypeOf(m.@) = BE \implies TypeOf(m.(@ + SCOffset)) = SCE$$

Definition 9.1.7 (PDTIfPDFlag)**.** When the PDflag of a Shadow 1 entry is set, it means the linked block in the Blocks structure hosts a PD structure.

$$\begin{aligned}
&\forall @, TypeOf(m.@) = SHE \wedge m.@.PDflag = true \implies \\
&TypeOf(m.(@ - Sh1Offset)) = BE \wedge (m.(@ - Sh1Offset)).accessible = true \\
&\wedge (m.(@ - Sh1offset)).present = true \\
&\wedge TypeOf(m.(@ - Sh1Offset).startaddress) = PD
\end{aligned}$$

Definition 9.1.8 (AccessibleNoPDFlag)**.** An accessible block cannot host a metadata structure.

$$\begin{aligned}
&\forall @, TypeOf(m.@) = BE \wedge m.@.accessible = true \\
\implies &TypeOf(m.(@ + SCoffset)) = SCE \wedge m.(@ + SCoffset).PDflag = false
\end{aligned}$$

Definition 9.1.9 (FirstFreeSlotPointerIsBEAndFreeSlot)**.** The reference to the first free slot has the type BE and is free.

$$\begin{aligned}
&\forall @, TypeOf(m.@) = PD \wedge m.@.firstfreeslot \not= NULL \\
\implies &TypeOf(m.@.firstfreeslot) = BE \wedge DEFAULT(m.@.firstfreeslot)
\end{aligned}$$

Definition 9.1.10 (BlocksRangeFromKernelStartIsBE)**.** Each block entry has the type BE. The $kernelStructureEntriesNb$ parameter bounds the index to an arbitrary value.

$$\begin{aligned}
&\forall @, \forall index, \\
&TypeOf(m.@) = BE \wedge m.@.blockindex = 0 \wedge index < kernelStructureEntriesNb \\
&\implies TypeOf(m.(@ + index)) = BE
\end{aligned}$$

Definition 9.1.11 (KernelStructureStartFromBlockEntryAddrIsKS)**.** The start of a super-

structure has the type BE.

$$\forall @, TypeOf(m.@) = BE \implies TypeOf(m.(@ - m.@.blockindex)) = BE$$

Definition 9.1.12 (sh1InChildLocationIsBE)**.** The reference to a block's location in the child partition has the type BE.

$$\forall @, TypeOf(m.@) = SH1E \implies TypeOf(m.@.inChildLocation) = BE$$

Definition 9.1.13 (StructurePointerIsKS)**.** The reference to the first superstructure is the start of a superstructure.

$$\forall @, TypeOf(m.@) = PD \implies (m.@.structure).blockindex = 0$$

Definition 9.1.14 (NextKSOffsetIsPADDR)**.** The value of the reference to the next linked superstructure has the type $PADDR$.

$$\begin{aligned}&\forall @,\\ &TypeOf(m.@) = BE \wedge (m.@).blockindex = 0\\ &\wedge TypeOf(m.(@ + NextOffset)) = PADDR\end{aligned}$$

Definition 9.1.15 (NextKSIsKS)**.** The reference to the next element of the linked list of superstructures is the start of another superstructure.

$$\begin{aligned}&\forall @,\\ &TypeOf(m.@) = BE \wedge (m.@).blockindex = 0 \wedge m.(@ + NextOffset)) \neq NULL\\ &\implies TypeOf(m.(@ + NextOffset)) = BE \wedge m.(@ + NextOffset).blockindex = 0\end{aligned}$$

Inner structural set properties

- NoDupInFreeSlotsList s
- freeSlotsListIsFreeSlot s
- DisjointFreeSlotsLists s
- inclFreeSlotsBlockEntries s

- DisjointKSEntries s
- noDupKSEntriesList s
- noDupMappedBlocksList s
- noDupUsedPaddrList s
- MPUInAccessibleBlocks s

These properties manipulate abstract structures, which could be implemented with lists or arrays. We speak about the free slots list of a partition *part* by the access $FreeSlotsList[part]$. We refer to the list of slots of a partition *part* with $Slots[part]$. The list of slots $Slots$ contains the free slots list $FreeSlotsList$ and the memory blocks $MappedBlocks$. We use the operator $NoDuplicate[list]$ stating all elements of *list* are unique.

Definition 9.1.16 (NoDupInFreeSlotsList)**.** Each element of a free slots list is unique.

$$\forall partition \in PartTree \implies NoDuplicate[FreeSlotsList[partition]]$$

Definition 9.1.17 (freeSlotsListIsFreeSlot)**.** Each element of a free slots list are free.

$$\forall partition \in PartTree, \forall @ \in FreeSlotsList[partition] \implies DEFAULT(m.@)$$

Definition 9.1.18 (DisjointFreeSlotsLists)**.** Given all partitions of a partition tree, all free slots lists are disjoint.

$$\begin{aligned}&\forall partition1 partition2 \in PartTree,\\ &\forall @a \in FreeSlotsList[partition1],\\ &\forall @b \in FreeSlotsList[partition2]\\ &\implies @a \neq @b\end{aligned}$$

Definition 9.1.19 (inclFreeSlotsBlockEntries)**.** The free slots list is included in the *Blocks* structure.

$$\begin{aligned}&\forall partition \in PartTree, \forall a \in FreeSlotsList[partition]\\ &\implies @a \in Blocks[partition]\end{aligned}$$

Definition 9.1.20 (DisjointKSEntries)**.** Given all partitions in a partition tree, all slots are unique.

$$\forall partition1 partition2 \in PartTree,$$
$$\forall @a \in Blocks[partition1], \forall @b \in Blocks[partition2] \implies @a \neq @b$$

Definition 9.1.21 (noDupKSEntriesList)**.** In a given partition, each slot is unique.

$$\forall partition \in PartTree \implies NoDuplicate[Blocks[partition]]$$

Definition 9.1.22 (noDupMappedBlocksList)**.** In a given partition, each mapped block is unique.

$$\forall partition \in PartTree \implies NoDuplicate[MappedBlocks[partition]]$$

Definition 9.1.23 (noDupUsedPaddrList)**.** In a given partition, no block overlaps another (the set of addresses they contain are disjoint).

$$\forall partition \in PartTree \implies NoDuplicate[AllPaddr[Blocks[partition]]]$$

Definition 9.1.24 (MPUInAccessibleBlocks)**.** In a given partition, all blocks configured in the MPU are accessible blocks belonging to that partition.

$$\forall partition \in PartTree, \forall block \in partition.MPU$$
$$\implies block \in AccessibleMappedBlocks[partition]$$

Inter-partition properties

- currentPartitionInPartitionsList s
- noDupPartitionTree s
- isParent s
- isChild s
- accessibleChildPaddrIsAccessibleIntoParent s
- sharedBlockPointsToChild s

These properties rule the relationship between partitions. $AllPaddr[blocks]$ refer to all addresses contained in the $blocks$.

Definition 9.1.25 (currentPartitionInPartitionsList)**.** The current partition belongs to the partition tree.

$$currentPartition \in PartTree$$

Definition 9.1.26 (noDupPartitionTree)**.** All partitions belonging to the partition tree are unique.

$$NoDuplicate[PartTree]$$

Definition 9.1.27 (isParent)**.** All children of a parent partition points to their parent.

$$\forall parentchild \in PartTree \land child \in children[parent] \implies child.parent = parent$$

Definition 9.1.28 (isChild)**.** All partitions pointing to the same parent are children of this parent.

$$\forall parentchild \in PartTree \land child.parent = parent \implies child \in children[parent]$$

Definition 9.1.29 (accessibleChildPaddrIsAccessibleIntoParent)**.** All accessible addresses in a partition (union of all addresses contained in the accessible mapped blocks) are mapped and accessible in their parent.

$$\forall parentchild \in PartTree,$$
$$\forall accessiblePaddr \in AllPaddr[AccessibleMappedBlocks[child]]$$
$$\implies accessiblePaddr \in AllPaddr[AccessibleMappedBlocks[parent]]$$

Definition 9.1.30 (sharedBlockPointsToChild)**.** Each block in a child partition has a corresponding block in the parent partition that contains the same addresses; block

which points to the child (in the Shadow 1 structure).

$$\forall parentchild \in PartTree,$$
$$\forall block \in MappedBlocks[child]],$$
$$\forall @ \in block,$$
$$\exists block' \in MappedBlocks[child]],$$
$$@ \in block' \wedge$$
$$((m.(block + Sh1Offset)).PDflag = true \vee (m.(block + Sh1Offset)).PDchild = child)$$

9.1.3 Definition of the isolation invariant

Definition 9.1.31 (Isolation invariant)**.** Pip-MPU's isolation invariant is a conjunct of the security properties S and the consistency properties C.

$$S \bigwedge C$$

To be noted, the isolation invariant is an umbrella term that hides that all properties do not participate to the isolation property, strictly speaking.

9.2 Proof methodology

Pip-MPU inherits Pip's proof methodology. In Pip, and so by extension Pip-MPU, the execution of kernel code is privileged. Proving the Kernel Isolation invariant implies the partitioning model can only be updated by privileged code, so by the kernel. Therefore, the system state is not modified during execution of userland partitions (recall that only kernel elements are modeled in the system state) and properties satisfied at the end of the execution of a service are the same when calling the next service later on. Thus, if the invariants are satisfied at the start of the execution of a service, we must prove they are still satisfied at the end of the execution. The invariants are only required to be true at start and end, but might have temporary states within the execution of a service where the invariants is not satisfied, for example during a metadata structure update. And we already know the invariants are satisfied for the first service to be called because Pip-MPU prepared the root partition to satisfy them. In a nutshell, we only need to prove the invariants at the end of a service execution starting from a system state that already satisfies them.

Transposing the previous paragraphs in Hoare logic means the following Hoare triple:

$\{VS\&HI\&KI\&C\}$
Pip-MPU service (*parameters*)
$\{VS\&HI\&KI\&C\}$

The Hoare triple concerns each service. A proof of a service can be realised independently from the proof of another service because the services cannot be preempted (exceptions are disabled, see Section 6.2.5). Thus, services are called one at a time, and Hoare triples are chained one after the other.

As there is no order in the service calls, the system state is unknown at the start of each service execution. In order to reason about the current system state, services all have a first **phase of memory checks** to uncover it and be assured the conditions are met to run the service and be able to prove the invariants at the end of the execution. Only then can the system state be updated in the hardware model.

9.3 Sketch of proof

From a functional point of view, Pip-MPU's services are distributed in two categories: pure informational services and modification services. We give the sketch of proof for the Pip-MPU services `findBlock` from the first category and `addMemoryBlock` from the second category. Because the proof target is Pip's security properties, we only demonstrate them by assuming the consistency properties hold. Obviously, the consistency properties must be proved in the full proof.

9.3.1 Proof of pure informational services: example with `findBlock`

This service looks for a block in the memory space of a partition and returns all its attributes. It takes as parameters a partition descriptor and any address contained in the searched block. `findBlock` has a first phase of checks to verify the user parameters are in their expected range and correspond to meaningful values: 1) the partition descriptor where to search for the block exists and is either the current partition or one of its children, and 2) the searched address is contained in one of the mapped blocks. Then, it reads information in memory and returns them to the user: it returns the block's attributes. The service's pseudo-code is given in Algorithm 1.

Algorithm 1 Pseudo-code of `FindBlock` (idPD: paddr) (addressInBlock : paddr)

```
1:  paramOK ← CheckParameters(idPD, addressInBlock)
2:  if paramOK IS FALSE then
3:    return NULL { parameters not OK, stop }
4:  else
5:    blockAddr ← findBlockInMemorySpace(IdPD, addressInBlock) { find the block }
6:    if blockAddr IS NULL { check the block exists } then
7:      return false { no block found, stop }
8:    else
9:      return block { Return the block }
10:   end if
11: end if
```

Proof goal: the isolation invariant.

$\{VS\&HI\&KI\&C\}$

`findBlock` *part addressInBlock*

$\{VS\&HI\&KI\&C\}$

Assumptions: the consistency properties C hold.

Proof.

The service does not modify the system state; thus the system state is the same as before the execution of the service. We know the invariants were satisfied at start, so they are still satisfied at the end of the service.

Qed.

We can observe that the sketch of proof is simple because of the unmodified system state.

9.3.2 Proof of modification services: example with `addMemoryBlock`

The `addMemoryBlock` service is interesting because it deals with a parent-child relationship, manipulates almost all memory types (except $PADDR$) and modifies a single partition, the child. `addMemoryBlock` also has a check phase of the user parameters before modifying the state. The order of modification instructions has no importance because we prove the security properties in the last step, when all modifications have passed. $s0$ refers to the initial state at service start while s' refers to a final state, whatever the path of execution. We write $list + +[element]$ to represent a list $list$ extended by the element $element$. `addMemoryBlock`'s pseudo-code is given in Algorithm 2 while

Figures 9.1 and 9.2 illustrate the local view on a parent partition's user space and one of its children respectively before and after the block sharing operation.

Algorithm 2 Pseudo-code of `addMemoryBlock` (idPDchild idBlockToShare: paddr) (r w e : bool)

```
1:  paramOK ← CheckParameters(idPDchild, idBlockToShare, r, w, e)
2:  if paramOK IS FALSE then
3:    return NULL { parameters not OK, stop }
4:  else
5:    if [idPDchild.nbFreeSlots <= 0
      or idPDChild.firstFreeSlotPointer IS NULL
      or blockToShareInCurrPart.accessible IS FALSE
      or blockToShareInCurrPart.present IS FALSE] then
6:      return NULL { no free slots left in child or first free slot pointer in child is null
        or block not accessible or block not present, stop }
7:    else
8:      shared ← CheckBlockAlreadySharedWithChild(idBlockToShare, idPDchild)

9:      if shared IS TRUE then
10:       return NULL { block shared, stop }
11:     else
12:       blockToShareChildEntry ← insertNewEntry(IdPDChild, idBlockToShare) {
          copy block in child }
13:       SetBlockSharedWithChild(idBlockToShare, idPDchild) { register the shared
          block in the parent}
14:       return blockToShareChildEntry { return the shared block slot in child }
15:     end if
16:   end if
17: end if
```

Proof goal: the isolation invariant.

$\{VS\&HI\&KI\&C\}$

`addMemoryBlock` *child blockToShare*

$\{VS\&HI\&KI\&C\}$

Assumptions: the consistency properties C hold.

Proof.

The proof has two sorts of exits: either the service is fed with incorrect user parameters (values not in expected range, block not existing or not accessible and present...)

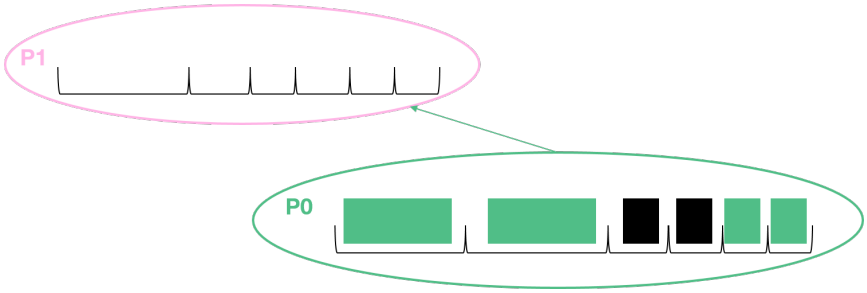


Figure 9.1: Local view on parent partition P0 and one of its children P1 *before* the call to the `addMemoryBlock` service. P1 does not have any blocks mapped in its memory space. P0 has many available blocks (black blocks hold the metadata structures of the child and are not available anymore).

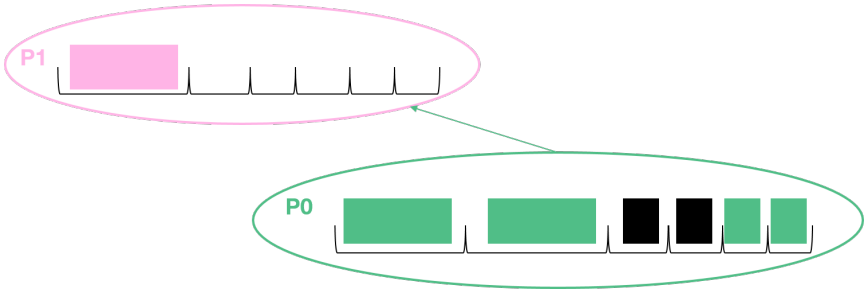


Figure 9.2: Local view on parent partition P0 and one of its children P1 *after* the call to the `addMemoryBlock` service. P1 got one block out of P0's available memory blocks.

and stops before modifying the state; or it passes through the whole modification phase. This shows the importance of not modifying the state before being sure the execution will go through the whole code to prevent exits that leave the state inconsistent with invariants that will fail.

Exit 1: incorrect user parameters Incorrect user parameters precipitate the execution path in returning the NULL address before any updates have been done. Because of no state modifications, the same sketch of proof as for pure informational services can be applied. *Qed.*

Exit 2: Correct user parameters With correct user parameters, the service executes to the last line. To be noted, the Vertical Sharing and the Horizontal Isolation properties consider combinations of several partitions with parent-child relationships that could be the parent-child relationship in `addMemoryBlock`. The $NoDupPartitionTree$ invariant guards against impossible cases implying cycles like the parent is also its own child. Furthermore, to assign the roles in the relationships, the $isChild$ and $isParent$ invariants support the proof.

Vertical Sharing The service does not modify the parent's mapped blocks, so $mappedBlocks[parent]_{s'} = mappedBlocks[parent]_{s0}$. On the contrary, the child's mapped blocks do change: $mappedBlocks[child]_{s'} = mappedBlocks[child]_{s0} + +blockToShare$.
We need to show that:
$\forall blockAddr \in AllPaddr[mappedBlocks[child]]_{s'}$,
$block \in AllPaddr[mappedBlocks[parent]]_{s0}$. So given an address $addr$, there are two cases:
1) $addr \in AllPaddr[mappedBlocks[child]]_{s0}$ or
2) $addr \in AllPaddr[blockToShare]$.

Case 1: $addr \in AllPaddr[mappedBlocks[child]]_{s0}$.
We know that the Vertical Sharing property is true at $s0$. *Qed.*

Case 2: $addr \in AllPaddr[blockToShare]$.
We know $blockToShare \in mappedBlocks[parent]_{s0}$ because we retrieve the block from that list. This means the $addr$ already was in one of the parent's blocks when the property was true at $s0$. *Qed.*

Horizontal Isolation For this property, the focus is set on two children of a certain parent partition. The children's memory spaces must not intersect (neither their mapped blocks nor their kernel structures). Two cases must be covered for this property: 1) one of the children is the *child* modified by the service `addMemoryBlock` (whether it is the first or the second child does not matter as the property is symmetric) and 2) the children are not *child*.

Case 1: The check phase states that the block is not shared at $s0$, which means we know that $blockToShare \notin mappedBlocks[child1]_{s0} \wedge$
$blockToShare \notin mappedBlocks[child2]_{s0}$. The service just modifies one of the children as explained in Vertical Sharing, leaving the other one untouched. We take here the example of $child1$. We get $mappedBlocks[child1]_{s'} = mappedBlocks[child1]_{s0} + +$ $blockToShare \wedge\ mappedBlocks[child2]_{s'} = mappedBlocks[child2]_{s0}$. For $child1$, there are then two options: either a block in the initial set of memory blocks $mappedBlocks[child1]_{s0}$ or it is $blockToShare$. Assuming the property is true at initial state $s0$ implies that the first option is trivial. Thus, remains the second option. Memory spaces are disjoint only if $blockToShare \notin mappedBlocks[child2]_{s'}$, which is equivalent to $blockToShare \notin$ $mappedBlocks[child2]_{s0}$ which follows from the check phase as stated above. *Qed.*

Case 2: for any partition not being the one modified in the service, their mapped blocks remain untouched in the modified state. Hence, $mappedBlocks[child1]_{s'} =$ $mappedBlocks[child1]_{s0} \wedge\ mappedBlocks[child2]_{s'} = mappedBlocks[child2]_{s0}$.

The property is then equivalent to the property which was satisfied at $s0$. *Qed.*

Kernel Data Isolation The current partition is not modified, which means the focus is set on the child partition *child*. Because the service calls only consider parent-child relationships, three cases must be covered here: 1) $partition1$ and $partition2$ are the same partition, 2) $partition1$ is $partition2$'s parent, and 3) $partition2$ is $partition1$'s parent. All cases are split in two possibilities: a) $partition1$ is the child in `addMemoryBlock` b) $partition2$ is the child in `addMemoryBlock`. All in all, six cases must be considered. However, the configuration blocks are not modified for any partition, not even the child in `addMemoryBlock`, so we need to prove:

$$\forall partition1, partition2 \in PartTree,$$
$$\forall blockAddr \in AllPaddr[AccessibleMappedBlocks[partition1]]_{s'},$$
$$blockAddr \notin AllPaddr[ConfigBlocks[partition2]]_{s0}.$$

This means the proof path is only crucial for changes in $partition1$, so possibility a). Indeed, in any other combination with possibility b), $partition1$ is different from the child in `addMemoryBlock` so its memory blocks are unchanged and so it leads to the initial state $s0$ ($AllPaddr[AccessibleMappedBlocks[partition1]]_{s'} = AllPaddr[AccessibleMappedBlocks[partition1]]_{s0}$) when the property is assumed true. *Qed.*

Case 1 a): $partition1$ and $partition2$ are the same partition, so the child in `addMemoryBlock`. An accessible memory block from the child is either one of the initial blocks or the newly copied block from the parent. Because the property is satisfied at the initial state, if it is one of the initial blocks, the proof is trivial. If the block is the newly added one, we know it is also part of the parent's memory space, especially at the initial state $s0$. Because there are no restrictions on relationships between $partition1$ and $partition2$ in the expression of the security property, the property is true at the initial state, especially for $partition1$ being the parent partition and $partition2$ being the child. The parent partition's accessible blocks did not change after service execution, so still the same set as at the initial state where the copied block belonged to this set. The property was satisfied at the initial state. *Qed.*

Case 2 a): with the child in `addMemoryBlock` being $partition1$, we know $AccessibleMappedBlocks[child]_{s'} = AccessibleMappedBlocks[child1]_{s0} ++ blockToShare$ because the new added block $blockToShare$ becomes accessible. Hence, two sub-cases: either we consider an address $addr$ in the initial accessible blocks or in $blockToShare$.

Furthermore, the service does not affect any partition except the current partition and the child in addMemoryBlock, so especially does not modify the eventual descendants of this child. Thus, the memory spaces of the children, notably for $partition2$, are unchanged compared to the initial state $s0$, and especially their configuration blocks: $ConfigBlocks[partition2]_{s'} = ConfigBlocks[partition2]_{s0}$

The first sub-case is then trivially verified because the security property is assumed true at $s0$ for the same combination of $partition1$ and $partition2$. *Qed.*

For the second sub-case, we need to consider another combination for the security property at $s0$, taking the parent (current) partition of addMemoryBlock and $partition2$, *i.e.*:

$addr \in AllPaddr[AccessibleMappedBlocks[current]]_{s0}$,
$addr \notin AllPaddr[ConfigBlocks[partition2]]_{s0}$.

Indeed, $blockToShare \in AccessibleMappedBlocks[current]]_{s0}$ because of the Vertical Sharing property and the fact that it is this block that is copied into the child $partition1$ during addMemoryBlock so it must be accessible as verified in the check phase of the service.

This means $addr$ also lies in one of the initial accessible blocks of the current partition when the property is assumed true. *Qed.*

Case 3 a): $partition2$ is the parent (current) partition of the child in addMemoryBlock. Similarly as discussed previously, the configuration blocks of the child and the accessible mapped blocks of the parent are untouched after executing the service, so $ConfigBlocks[partition2]_{s'} = ConfigBlocks[partition2]_{s0} \wedge$
$AllPaddr[AccessibleMappedBlocks[current]]_{s'} =$
$AllPaddr[AccessibleMappedBlocks[current]]_{s0}$. This is equivalent to proving the property at $s0$, which is assumed to be true. *Qed.*

9.4 Model and implementation in Coq

In this section, we introduce the abstract model on which the proofs are conducted. Nevertheless, the abstract level is still a *concrete model* because it is located at the same implementation level as the services that are literally translated in the C language.

In this section, and the rest of the document, Gallina is used for examples involving the model while C is used for examples concerning the C implementation. We use the bottom-up approach to present the model with corresponding implementation in a

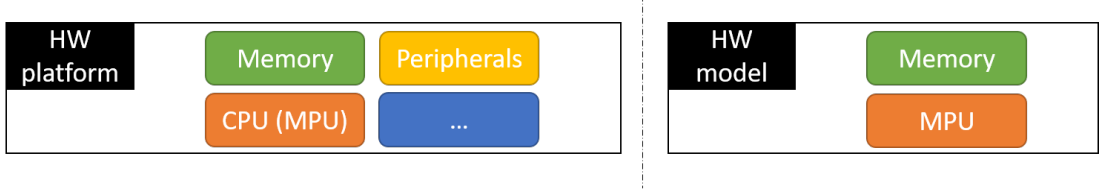


Figure 9.3: On the left, the real hardware composed of many elements. On the right, the abstract hardware model used in the proofs composed of the memory model and the MPU model.

second phase.

9.4.1 Basic abstract types

Our model defines several basic types.

paddr represents a physical address in the memory. It is bounded by the constant *maxAddr*.

```
Record paddr := {
  p :> nat;
  Hp : p <= maxAddr }.
```

index represents a natural number. It is bounded by the constant *maxIdx*.

```
Record index := {
  i :> nat ;
  Hi : i <= maxIdx }.
```

9.4.2 Hardware Abstraction Layer (HAL)

Formal model of the hardware

As far as we are concerned, Pip's security properties must be verified on a hardware platform composed of a CPU, MPU, memory, peripherals, and any other hardware components. However, the properties only depend on the kernel's effects, *i.e.* how Pip-MPU interacts with the hardware platform. This means the abstracted view does not need to include all the hardware platform's components because Pip-MPU just manipulates kernel structures and the MPU. Thus, the **abstracted hardware model** is just composed of a **memory model** and an **MPU model** as shown in Figure 9.3. In particular, it does not include registers nor the current privilege mode [29] because we only consider Pip-MPU services that always execute in privileged mode.

Formal model of the memory The memory is modeled as an association list (list of key-value pairs). It associates a physical address with a value, just like Pip (MMU)'s hardware model. The value is either a **kernel structure element** (see below their definitions) or another physical address (representing a pointer). Indeed, the memory model is only filled with kernel elements because the services execute in privileged mode, reserved for the kernel, and as such never include userland operations. Furthermore, Pip's security properties only concern kernel structures and never the contents of the memory blocks of the userland partitions. Modeling these contents would then be useless to prove the properties.

Formal model of the Memory Protection Unit The MPU configuration is modeled as a list of block identifiers (the active memory blocks of the current partition).

```
MPU : list paddr ;
HMPU : length(MPU) <= MPURegionsNb
```

This is a very simplified view of an MPU. It does not reflect the hardware constraints on the MPU region (size and alignments). The rationale is that Pip-MPU should be portable on different MPU versions and architectures that might have different constraints. The constraints are taken into account in the C implementation of the HAL. The adapted behavior for the considered MPU is customised manually. The partial reconfiguration feature on ARMv7-M MPUs introduced in Section 6.4.4 is an example.

Formal model of the system state

The system state captures the evolution of the system at a certain instant. Our system state is the same as Pip's. It considers a **system state** composed of the **memory state** and the **current partition state**. The current partition state is the identifier of a partition. This is the partition that calls the service, i.e. the parent partition for services operating on a child. The memory model has been introduced above. The state is updated each time a write operation modifies the memory model or the current partition identifier.

```
Record state : Type := {
 currentPartition : paddr;
 memory : list (paddr * value)
}.
```

Note that the MPU model is not present in the system state. Indeed, the MPU configuration depends on the partition and is stored in the PD kernel structure. As such, the MPU model is stored in the memory model and the current partition identifier is enough to find the PD structure and retrieve the MPU state.

Low-level primitives and the monad

Low-level primitives read and write from memory. Pip uses a monad to withhold the side effects of the code in a simpler form. The low-level primitives make use of the monad to interact with the hardware model.

The monad Pip-MPU is programmed in Gallina, a functional language, which is not easily readable for developers more acquainted with the imperative style like C. Purely functional languages do not capture the side effects of functions in global or local variables like imperative languages do. To bring out the side effects, Pip uses a monadic programming style which represents computations in an abstract data type, the monad. The monad style is better suited for formal verification using the Hoare logic 8.2.1. The monad encapsulates a state and provides functions to access it. Hence, the monad transforms the functional code into a more C-like structure which eases the word-by-word translation of the code from Gallina to C in the Pip workflow. Our model integrates the same monad as in Pip (MMU). Pip's monad is composed of the monad type constructor *LLI* and the functions *put*, *get*, *ret* and *bind*.

The *LLI* constructor wraps the system state in a *result* type (the monadic type). The *result* associates any other type with the state.

```
Definition LLI (A :Type) : Type := state -> result (A * state).

Inductive result (A : Type) : Type :=
| val : A -> result A
| undef : nat -> state -> result A.
```

The *put*, *get*, *ret* constructors manipulate the state and return a new monad. *put* modifies the state monad. *get* retrieves the state monad. *ret* gives the value of a computation, so returns a state monad from an object of any type. *bind* composes a monadic function with other monadic functions, which gives the sequential/imperative style to the code by chaining functions one after the other.

```
Definition put (s : state) : LLI unit :=
  fun _ => val (tt, s).

Definition get : LLI state :=
  fun s => val (s, s).

Definition ret {A : Type} (a : A) : LLI A :=
```

```
  fun s => val (a , s) .

Definition bind {A B : Type} (m : LLI A)(f : A -> LLI B) : LLI B :=
  fun s => match m s with
         | val (a, s') => f a s'
         | undef a s' => undef a s'
         end.
```

The *A* and *B* types roughly correspond to variables and arguments of functions in the imperative style. We also use the same Pip notation $perform\ x := m\ in\ e$ which indicates the use of *bind* between *m* and *e*, that means the sequential instructions of *m* and *e* with *e* depending on the value *x* returned by *m*. Likewise, we use the notation $m1;;m2$ to translate a *bind* between $m1$ and $m2$ that does not depend on the returned value of $m1$.

modify applies a function to the state monad. It simulates the interaction with the real hardware.

```
Definition modify (f : state ! state) : LLI unit :=
  perform s := get in put (f s).
```

Low-level primitives The low-level primitives heavily interact with the state monad to develop the system state. They define all reads and writes to the memory model (which includes the MPU model, as stated above).

For example, the function *readPDTable* in Listing 1 retrieves the structure PD at address *addr* in the memory model. It first retrieves the state, looks into the memory to find the physical address *addr* and either returns a monad from the structure of type PDT at that memory address if the correct type matches, and if not is an undefined behavior with an error code.

Listing 1 Definition of `readPDTable` in Coq.

```
Definition readPDTable (addr : paddr)  : LLI PDTable :=
  perform s := get in (*retrieve state*)
  let entry := lookup addr s.(memory) beqAddr in (*find entry at address
   ↪  addr*)
  match entry with
  | Some (PDT a) => ret a (*entry matches the PDT type -> OK, return
   ↪  structure*)
  | Some _ => undefined 5 (*no match -> NOK, undefined behavior*)
  | None => undefined 4 (*no match -> NOK, undefined behavior*)
  end.
```

9.4.3 Formal model of the kernel structures

As proofs are at implementation level, the model uses the same kernel structures presented earlier in chapter 7. Recall that kernel structures may be of two types: PD and superstructure. They are formalised in Gallina as records.

PD structure *PDTable* is the type of a PD structure. It holds all previously discussed fields. The field *vidtBlock* is used by the `yield` service relating to context switch and exceptions and thus not discussed further in this dissertation. Note the presence of the MPU model presented above. It is stored in the PD kernel structure as a list of block identifiers (the active memory blocks).

```
Record PDTable :=
{
 structure : paddr ;
 firstfreeslot : paddr ;
 nbfreeslots : index ;
 nbprepare : index ;
 parent : paddr ;
 MPU : list paddr ;
 vidtBlock : paddr ;
 nbprepare < maxNbPrepare
}
```

Note the constraints on the *nbprepare* which limits the number of prepare of a partition to some user-defined value.

Superstructure Superstructures are not modeled as a single type. Instead, we model each of the superstructure entry independently, which means each of the internal entries of the structure *virtual* MPU, Shadow 1 and Shadow Cut. Again, the entry fields correspond to their respective description in the first part.

```
(* **virtual** MPU entry type *)
Record BlockEntry : Type:=
{
 read : bool; (*read right*)
 write : bool ; (*write right*)
 exec : bool; (*execution right*)
 present : bool; (*present flag*)
 accessible : bool; (*accessible flag*)
```

```
 blockindex : index; (* index of the entry in the structure*)
 blockrange : block ; (*block start and end addresses*)
 Hidx : blockindex < kernelStructureEntriesNb
}.

(* block represents a contiguous region of memory addresses *)
Record block := {
  startAddr : paddr; (*block start address*)
  endAddr : paddr ; (*block end address*)
  Haddr : startAddr < endAddr ;
}.

(* Shadow 1 entry type *)
Record Sh1Entry : Type:=
{
 PDchild : paddr; (*child identifier*)
 PDflag : bool; (*flag indicating the related block's content holds a PD
  ↪  structure*)
 inChildLocation : paddr (*block initial copy address in the child*)
}.

(* Shadow Cut structure entry type *)
Record SCEntry : Type:=
{
 origin : paddr; (*address of the initial block before any cut*)
 next : paddr (*pointer to the next subblock*)
}.
```

Note the constraints on the *blockindex* field which should not exceed a certain user-defined value and the constraints on the block's start address which must be lower than the same block's end address.

Furthermore, we introduce now the *value* constructor. The type *value* is used in the memory model as the *value* in the key-value pairs. In other words, each memory address can only be tagged with the following types:

```
Inductive value : Type:=
|BE : BlockEntry -> value
|SHE : Sh1Entry -> value
|SCE : SCEntry -> value
```

```
|PDT : PDTable -> value
|PADDR : paddr -> value.
```

We clearly observe the memory model is only composed of kernel structure elements or physical addresses as stated in 9.4.2. The consistency of the superstructure is ensured by consistency properties that are presented in the next chapter.

9.4.4 Formal model of user defined values

The user has the possibility to define some fixed values that are used at compile-time. These values correspond to the number of entries (indexes) in a kernel structure and to the maximum number of prepare which practically bounds the number of memory blocks in a memory space by the formula disclosed in Section 7.4.4.

```
Definition kernelStructureEntriesBits := 3.
Definition nbPrepareMaxBits := 3.
Definition kernelStructureEntriesNb := kernelStructureEntriesBits ^ 2.
Definition maxNbPrepare := nbPrepareMaxBits ^ 2.
```

These are arbitrary numbers. Their content does not influence the proofs; however the presence of the bounds implies to demonstrate these properties when constructing or modifying the type.

9.4.5 Formal expression of the security properties

We define the security properties with the Coq formalism below. The formal expression of Pip's security properties in Pip-MPU mirrors their expression in Pip (MMU) (see later Section 11.1.3 for comparison).

```
(** THE VERTICAL SHARING PROPERTY:
    All used physical addresses of a random child
    stem from the parent partition  *)
Definition verticalSharing s : Prop :=
  forall parent child : paddr,
  In parent (getPartitions multiplexer s) ->
  In child (getChildren parent s) ->
  incl (getUsedPaddr child s) (getMappedPaddr parent s).


(** THE ISOLATION PROPERTY BETWEEN PARTITIONS,
    Two different children of the same parent
```

```
    have no used physical addresses in common.*)
Definition partitionsIsolation  s : Prop :=
  forall parent child1 child2 : paddr ,
  In parent (getPartitions multiplexer s) ->
  In child1 (getChildren parent s) ->
  In child2 (getChildren parent s) ->
  child1 <> child2 ->
  disjoint (getUsedPaddr child1 s) (getUsedPaddr child2 s).


(** THE ISOLATION PROPERTY BETWEEN THE KERNEL DATA AND PARTITIONS
   No metadata structure (physical addresses within them)
    is accessible from another partition. *)
Definition kernelDataIsolation s : Prop :=
      forall partition1 partition2 : paddr,
      In partition1 (getPartitions multiplexer s) ->
   In partition2 (getPartitions multiplexer s) ->
      disjoint (getAccessibleMappedPaddr partition1 s) (getConfigPaddr
       ↪  partition2 s).
```

The **used** addresses of a partition is the concatenation of all addresses contained in its mapped blocks (*i.e.* the *MappedPaddr*) and in its kernel structures (*i.e.* the *ConfigPaddr* of the ConfigBlocks):

```
1  Definition getUsedPaddr (partition : paddr) s : list paddr :=
2  let ksList := getConfigPaddr partition s in
3  let mappedblockList := getMappedPaddr partition s in
4  ksList ++ mappedblockList.
```

9.5 Properties propagation and proof levels

All the services use the low-level primitives, modeled as previously described, which interact with the memory model to read or write values in it. Because they are the basic blocks of the services, they create the context on which more abstract properties are proven, such as the security properties. We discuss in this section how and why some properties are constructed and propagated.

There are two levels of proofs for each low-level primitive. The first level extracts raw information from the memory model and propagates properties. The second level also propagates these properties but also constructs more elaborated properties that will be used by more abstract levels.

Propagated and elaborated properties out of low-level primitives differ again for read-only primitives and for modification primitives.

9.5.1 Read-only primitive: propagate all properties

Read-only primitives give back information located at some address in the memory model. Because they do not modify the state, they propagate all the properties of the current state.

We take the example of `readBlockEntryFromBlockEntryAddr` which reads the memory at some address *paddr* and returns the block entry at that address.

```
Definition readBlockEntryFromBlockEntryAddr (paddr : paddr) : LLI
↪  BlockEntry :=
 perform s := get in
 let entry := lookup paddr s.(memory) beqAddr in
 match entry with
 | Some (BE a) => ret a
 | Some _ => undefined 12
 | None => undefined 11
 end.
```

The first-level proof of `readBlockEntryFromBlockEntryAddr` states the minimum condition of success to get a valid result from the memory model (the *weakest precondition*, see Section 8.2.1). We observe in the primitive three possible exits: one `ret` and two `undefined`. The two exits respectively, represent the success and the failure paths of the primitive.

The difference between the exits lies in the precondition: either the memory holds the expected entry (`ret`) or the proof context is false (`undefined`). In a valid case, the path ending with a `ret`, properties satisfied before the operation are propagated after. In the false case, so a path ending in an `undefined` behavior, any properties are satisfied because anything can be proven from a false hypothesis. Then, the first-level proof must assume the value is in memory to end in the valid case, as illustrated in Listing 4, or must find a contradiction during the proof which corresponds to ending in the `undefined` path.

Note that the post-condition do not take into account the `undefined` path. Indeed,

Listing 2 Proof of `ret`

```
Lemma ret  (A : Type) (a : A) (P : A -> state -> Prop) :
{{ P a }}
ret a
{{ P }}.
Proof.
intros s H; trivial.
Qed.
```

Listing 3 Proof of `undefined`

```
Lemma undefined  (A : Type)(a : nat) (P : A -> state -> Prop) :
{{ fun s => False }}
undefined a
{{ P }}.
Proof.
intros s H; trivial.
Qed.
```

Listing 4 First-level lemma of `readBlockEntryFromBlockEntryAddr` (proof hidden for clarity)

```
Lemma readBlockEntryFromBlockEntryAddr  (addr : paddr) (P : BlockEntry ->
↪  state -> Prop) :
{{fun s  =>  exists addrentry : BlockEntry, lookup addr s.(memory) beqAddr
↪  = Some (BE addrentry)
          /\ P addrentry s }}
MAL.readBlockEntryFromBlockEntryAddr addr
{{P}}.
```

the Hoare triple holds only if the primitive terminates, which is not the case with undefined behaviours. Such a path would reveal a contradiction in the proof context. Thus, every lemma we will encounter proves the absence of undefined behaviours.

The second-level proof uses more abstract properties which connect to the previous preconditions, as illustrated in Listing 5. The expected type of the value in memory is captured in the `isBE` property and the post-condition states that all properties in the precondition are propagated, that the type of the value at that address has not changed with `isBE` and propagates its value through the property `entryBE`.

```
Definition isBE paddr s: Prop :=
match lookup paddr s.(memory) beqAddr with
|Some (BE _) => True
|_ => False
end.
```

```
Definition entryBE entryaddr be s:=
match lookup entryaddr s.(memory) beqAddr with
| Some (BE entry) => be = entry
| _ => False
end.
```

Modification primitive: partial propagation and new properties

Similarly, there are two levels of proofs for modification primitives. However, the difference here is that the post-condition might not propagate the same properties as in the precondition because the memory state has changed. We illustrate this case with `writeSh1InChildLocationFromBlockEntryAddr`.

Listing 5 Second-level lemma of `readBlockEntryFromBlockEntryAddr` (proof hidden for clarity)

```
Lemma readBlockEntryFromBlockEntryAddr (paddr : paddr) (P : state -> Prop)
 ↪  :
{{ fun s => P s /\ isBE paddr s  }}
MAL.readBlockEntryFromBlockEntryAddr paddr
{{ fun (be : BlockEntry) (s : state) => P s /\ isBE paddr s /\ entryBE paddr
 ↪  be s }}.
```

Primary considerations This example is more elaborated than the previous one because it writes in a Shadow1 field which is referenced via a block entry, *i.e.* located at the offset `sh1offset` from that block entry. The write operation is then composed of a first access to compute the offset (`getSh1EntryAddrFromBlockEntryAddr`) followed by the write itself at that offset address (`writeSh1InChildLocationFromBlockEntryAddr2`).

```
Definition writeSh1InChildLocationFromBlockEntryAddr (blockentryaddr :
↪  paddr) (newinchildlocation : paddr) : LLI unit :=
    perform Sh1EAddr := getSh1EntryAddrFromBlockEntryAddr
    ↪  blockentryaddr in
    writeSh1InChildLocationFromBlockEntryAddr2 Sh1EAddr
    ↪  newinchildlocation.
```

In the previous example, we had a direct access to the entries in the memory model, which is the case of the second operation `writeSh1InChildLocationFromBlockEntryAddr2` but not of `getSh1EntryAddrFromBlockEntryAddr` composed of different low-level primitives with direct access.

```
Definition getSh1EntryAddrFromBlockEntryAddr (blockentryaddr : paddr) :
↪  LLI paddr :=
    perform BlockEntryIndex := readBlockIndexFromBlockEntryAddr
    ↪  blockentryaddr in
    perform kernelStartAddr := getKernelStructureStartAddr
    ↪  blockentryaddr BlockEntryIndex in
    perform SHEAddr := getSh1EntryAddrFromKernelStructureStart
    ↪  kernelStartAddr BlockEntryIndex in
    ret SHEAddr.
```

This shows each operation is eventually defined of several direct access read and write operations. The first-level proof is conducted on them, one for each operation.

First-level proof We demonstrate here the first-level proof of `writeSh1InChildLocationFromBlockEntryAddr2` considering the elements of `getSh1EntryAddrFromKernelStructureStart` are proven in a similar manner for direct write operations or like the previous example for read operations.

Again, we observe three exits to the primitive: one valid exit ending with the modification of an entry in the memory model and two exits that went wrong with `undefined` behaviours. The *modify* operation here changes the current state s by adding

```
Definition writeSh1InChildLocationFromBlockEntryAddr2       (Sh1EAddr :
↪  paddr) (newinchildlocation : paddr) : LLI unit :=
 perform s := get in
 let entry := lookup Sh1EAddr s.(memory) beqAddr in
 match entry with
 | Some (SHE a) => let newEntry := {|
  PDchild := a.(PDchild);
  PDflag := a.(PDflag);
  inChildLocation := newinchildlocation
 |} in
  modify (fun s => {|
    currentPartition := s.(currentPartition);
    memory := add Sh1EAddr (SHE newEntry) s.(memory) beqAddr
 |})
 | Some _ => undefined 12
 | None => undefined 11
 end.
```

a new value at address $Sh1EAddr$. It *adds* a new value in the memory model because all modifications to the state are recorded as modification to the initial state. A direct read access at this address would retrieve the current value at that address, so the last value recorded.

The first-level proof is different from the previous read example from that not only types but also contents of entries are important. Indeed, direct write operations must assume a correct entry type to write a value in one of the entry's fields. It should also include properties that are satisfied after the change. Therefore, the precondition must demonstrate the correct entry type and mention properties that satisfy the new state. If the condition is met, then all satisfied properties are propagated after the operation, as illustrated in Listing 6.

Second-level proof The second-level proof concatenates preconditions necessary to perform `getSh1EntryAddrFromKernelStructureStart` and `writeSh1InChildLocationFromBlockEntryAddr2` and eventually deduces new properties.

Likewise, `getSh1EntryAddrFromKernelStructureStart`'s second-level proof concatenates all preconditions of the first-level proof of its direct access operations. The corresponding Hoare triple detailed in Listing 7 illustrates this and shows the preconditions to dispose of a *BE* type at some address, any properties satisfied at the current state s and inner partition purely structural invariants that call a proof that the computed offset is correct. Because the primitive does not modify the state, we recognise a similar

Listing 6 First-level lemma of `writeSh1InChildLocationFromBlockEntryAddr2` (proof hidden for clarity)

```
Lemma writeSh1InChildLocationFromBlockEntryAddr2 (Sh1EAddr
 ↪ newinchildlocation : paddr) (P : unit -> state -> Prop) :
{{fun s => exists entry , lookup Sh1EAddr s.(memory) beqAddr = Some (SHE
 ↪ entry) /\
P tt {|
  currentPartition := currentPartition s;
  memory := add Sh1EAddr
            (SHE {| PDchild := entry.(PDchild);
                    PDflag := entry.(PDflag);
                    inChildLocation := newinchildlocation |})
            (memory s) beqAddr |} }}
MAL.writeSh1InChildLocationFromBlockEntryAddr2 Sh1EAddr
 ↪ newinchildlocation {{P}}.
```

post-condition to the previous example with a propagation of the satisfied properties, the computed offset in property `sh1entryAddr`, and related Shadow 1 type *SHE*.

The second-level proof of `writeSh1InChildLocationFromBlockEntryAddr` groups all preconditions and post-conditions as illustrated in Listing 8.

9.6 C implementation equivalents

This section links the Coq implementation with the C implementation and the real hardware. The reader can directly compare the equivalent C types and data structures with the model presented above 9.4.

9.6.1 Basic abstract types

In C, *paddr* is equivalent to the `void*` datatype.

```
/* Paddr */
typedef void* paddr;
```

maxAddr is then the maximum value of this datatype. Dereferencing a pointer of type *paddr* with value 0 corresponds to NULL in C.

In C, *index* is equivalent to the uint32_t datatype. *maxIdx* is then the maximum value of this datatype.

Listing 7 Invariant of getSh1EntryAddrFromBlockEntryAddr

```
Lemma getSh1EntryAddrFromBlockEntryAddr  (blockentryaddr : paddr) (Q :
 ↪  state -> Prop) :
{{fun s =>  exists entry, lookup blockentryaddr s.(memory) beqAddr = Some
 ↪  (BE entry)
   /\ Q s
/\ wellFormedFstShadowIfBlockEntry s
/\      KernelStructureStartFromBlockEntryAddrIsKS s}}
MAL.getSh1EntryAddrFromBlockEntryAddr blockentryaddr
{{ fun sh1entryaddr s => Q s /\ sh1entryAddr blockentryaddr sh1entryaddr s
 ↪  /\ exists sh1entry : Sh1Entry,
lookup sh1entryaddr s.(memory) beqAddr = Some (SHE sh1entry)
}}.
```

Listing 8 Second-level lemma of writeSh1InChildLocationFromBlockEntryAddr (proof hidden for clarity)

```
Lemma writeSh1InChildLocationFromBlockEntryAddr (blockentryaddr
 ↪  newinchildlocation : paddr)  (P : unit -> state -> Prop) :
{{fun  s => exists entry , lookup (CPaddr (blockentryaddr + sh1offset))
 ↪  s.(memory) beqAddr = Some (SHE entry) /\
P tt {|
  currentPartition := currentPartition s;
  memory := add (CPaddr (blockentryaddr + sh1offset))
            (SHE {|       PDchild := entry.(PDchild);
                      PDflag := entry.(PDflag);
                      inChildLocation := newinchildlocation |})
            (memory s) beqAddr |}
/\ isBE blockentryaddr s
/\ wellFormedFstShadowIfBlockEntry s
/\ KernelStructureStartFromBlockEntryAddrIsKS s
 }}
MAL.writeSh1InChildLocationFromBlockEntryAddr blockentryaddr
 ↪  newinchildlocation  {{P}}.
```

9.6.2 Hardware Abstraction Layer

Figure 8.1 shows that only the low-level primitives are passed on, since we use the real hardware platform and do not need the hardware model anymore. Furthermore, the monad has no sense in the C implementation. This is the reason the C equivalents of the HAL must be written manually, to get rid of the monad.

9.6.3 System state

As the real hardware is used, the memory model is useless. Only the current partition identifier is passed on to the C implementation. It is declared as a global variable and updated at each context switch.

```
paddr current_partition; /* Current partition, default root */
```

9.6.4 Kernel structures

Each kernel structure in Gallina has a close C equivalent. This ensures that the services, on which the proofs are conducted, manipulate equivalent kernel structures in the implementation model than in the C implementation.

For example, the equivalent C implementation of the `PDTable` type is the structure:

```
/**
 * \struct PDTable
 * \brief PDTable structure
 */
typedef struct PDTable
{
    struct KStructure *structure      ;  //!< Pointer to the first
↪  kernel structure of the structure linked list
    void* firstfreeslot               ;  //!< Pointer to the first free
↪  slot in one of the kernel structures (if any)
    uint32_t nbfreeslots              ;  //!< Number of free slots left
    uint32_t nbprepare                ;  //!< Number of Prepare done on
↪  this partition
    struct PDTable *parent            ;  //!< Pointer to the parent
↪  partition
    BlockEntry_t *mpu[MPU_REGIONS_NB]  ;  //!< List of pointers to
↪  enabled blocks
    uint32_t LUT[MPU_REGIONS_NB*2]    ;  //!< MPU registers'
↪  configuration sequence
```

```
    BlockEntry_t *vidtBlock          ;  //!< Pointer to the block
↪  containing the VIDT.
} PDTable_t;
```

Note that the field constraints have disappeared because they only have sense during the proofs while in the C implementation we already use the proven services. Also, we can observe the heterogeneous nature of the pointers. Indeed, their types are resolved in the proofs with properties while in the C implementation we directly identify the target structure. Equivalently, we could have used the *void** type for all pointers and resolve the structure type when dereferencing it. Moreover, we note the additional presence of the array LUT. The latter seconds the MPU and represents the register pairs to be slammed in the MPU configuration at context switch (*cf.* Section 7.4.3). It only has a sense for the implementation and is thus transparent to our model.

9.6.5 Low-level primitives

The C low-level primitives modify the hardware as the primitives in the model equivalently modify the hardware model.

For example, the equivalent primitive of `readPDTable` presented earlier in Listing 1 is illustrated in Listing 9.

Listing 9 C equivalent definition of `readPDTable` defined in Coq in Listing 1.

```
/*!
 * \fn PDTable_t readPDTable(paddr pdaddr)
 * \brief Gets the Partition Descriptor (PD).
 * \param pdaddr The address where to find PD
 * \return the PD table
 */
PDTable_t readPDTable(paddr pdaddr)
{
    // Cast it into a PDTable_t structure
    PDTable_t* pd = (PDTable_t*)pdaddr; // TODO: Exception ? Only
↪  called with current partition

    // Return the pd table
    return *pd;
}
```

9.6.6 User-defined values

User-defined values are fixed at compile-time. For this, we use `defines` as defined in Listing 10.

Listing 10 C `defines` of user-defined values.

```
#define KERNELSTRUCTUREENTRIESBITS 3
#define NBPREPAREMAXBITS 3
#define KERNELSTRUCTUREENTRIESNB (1<<KERNELSTRUCTUREENTRIESBITS) //!<
↪  The number of entries in a kernel structure.
#define MAXNBPREPARE (1<<NBPREPAREMAXBITS) //!< The maximum number of
↪  times a partition can be prepared.
```

9.7 Discussion and limitations

Our hardware model is simplistic because the proofs abstract away the real hardware. Indeed, the formal HAL models low-level operations while it is manually rewritten in C to be portable on similar hardware. The formal and C-implemented HAL, as well as the services written in Coq and translated C code, show very close similarities that demonstrate the smooth surface between the proof and the executable workflow. The C-HAL also consists of optimisation and architecture-specific primitives like the partial MPU reconfiguration due to ARMv7-M hardware constraints. This sets the limits of our model and the reasoning basis for the proofs.

Furthermore, formal specification of the target hardware is not available. We had to create a formal model of the MPU in the shape of the active blocks list. Unfortunately, this is far from the ideal situation where the hardware already formally presents its specifications. Recent efforts are heading to this direction with the ARMv8 architecture [147] (not available for Cortex-M devices though). A correct specification of the hardware is a strong assumption for our proofs.

Moreover, we do not model the CPU in our hardware model. The information about the CPU mode (privileged/unprivileged) is not required because the services always run privileged and we do not need to consider the partitions behavior and content to prove the security properties. Registers are also not modeled because we are only interested in the holistic view of the partitioning scheme and not such low-level details. This means memory leaks on registers are out of scope of this work. We do not model caches as well for the same reason and because they are usually implementation-defined, many times absent and have parameterised countermeasures like cache flushing.

9.8 Conclusion

In this chapter, we presented the proof framework and the formal proof model in Coq for the upcoming formal proof of Pip's security properties in the next chapter. The proof goal is to prove the isolation invariant composed of the security properties and invariants on the Pip-MPU data structures. We discussed proof levels which extract properties and combine properties from low-level primitives. The formal model is enough for the proof workflow. We also presented the C implementation of the data structures and low-level primitives to show close equivalence with the formal model. Because the services are translated directly from the Coq model via the custom tool Digger, we have also set up the executable workflow. All in all, this chapter closes with a complete functional Pip workflow to produce proofs and an executable inheriting the proof properties. The proofs are conducted directly at implementation level, not using refinement techniques like many formally verified state-of-the-art systems.

Chapter 10

Formal proof of the security properties

Chapter outline

In this chapter, we introduce the formal verification of Pip-MPU. The process consists in proving that Pip's security properties hold for Pip-MPU's services. We demonstrate the formal proofs on Pip-MPU's `findBlock` and `addMemoryBlock` services via the Coq Proof Assistant.

As earlier, we only present the proof of the security properties given consistency

properties serve as a basis for the security properties which are the target verification. The curious reader finds all proofs and internal lemmas online [1], including the proofs of the consistency properties.

While independent, this chapter connects to all chapters of this second part. The formal verification follows Pip's proof workflow and the sketch of proof described in the previous chapter. It applies the proof techniques described in the next chapter. And the results, *i.e.* the proofs, are evaluated with elements discussed in Chapter 12. Finally, it is the raw results of a learning path and fine adjustments presented in the last chapter.

Fast reading path: Technical knowledge and know-how on Coq is not mandatory for this part because Coq's reasoning language is verbose but recommended for easier reading. Otherwise, the reader might want to skip the technical details and directly engage in the sections 10.3 and 10.4 respectively, on the proof metrics and discussions.

10.1 Proof of findBlock

The full proof is accessible online [2].

The proof goal is:

```
Lemma findBlock (idPD : paddr) (addrInBlock : paddr) :
{{fun s => partitionsIsolation s /\ kernelDataIsolation s /\
↪  verticalSharing s /\ consistency s }}
Services.findBlock idPD addrInBlock
{{fun _ s  => partitionsIsolation s /\ kernelDataIsolation s /\
↪  verticalSharing s /\ consistency s }}.
```

We said previously that the `findBlock` system call is a purely informational service, *i.e.* with no state modification. Our previous intuition in Section 9.3.1 indicated that we only had to propagate the properties of the initial state to the last instruction. The properties true in the precondition, in particular the security properties, are logically still true after the service's execution if the state has not changed.

10.1.1 Proof context

With Hoare logic, properties are unveiled instruction by instruction. We take the example of the penultimate instruction.

```
perform blockentry := readBlockEntryFromBlockEntryAddr blockAddr in
```

[1] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/addMemoryBlock_proof/proof/invariants

[2] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/blob/addMemoryBlock_proof/proof/invariants/FindBlock.v

Listing 11 Proof of `readBlockEntryFromBlockEntryAddr`

```
1  eapply bindRev.
2  { (** MAL.readBlockEntryFromBlockEntryAddr *)
3        eapply weaken. apply readBlockEntryFromBlockEntryAddr.
4        intros. simpl. split. apply H1. intuition.
5        - (* blockAddr = nullAddr, this is false since we are in the
         ↪  branch where the block is found *)
6              contradict H11. apply beqAddrFalse in H3. congruence.
7        - (* blockAddr <> nullAddr*)
8          unfold isBE. destruct H12. rewrite H6;trivial
9  }
```

The goal for this instruction proof is to propagate the properties racked up until that instruction. The proof context is then composed by all accumulated properties from the previous instructions. The Coq proof part addressing that is transcribed next in Listing 11.

This instruction is a low-level primitive and the properties it propagates have been discussed in the last Chapter 5. Lines 1 to 3 of the instruction proof in Listing 11 respectively, isolate the instruction from the rest of the instructions in the service, *weaken* the proof by requiring only the minimal set of properties, and finally apply the primitive's second-level proof.

The second-level proof requires two properties as precondition $P\ s \wedge isBEpaddr\ s$. This precondition is separated into two goals by the *split* tactic. The first element requires the properties to propagate. The hypothesis H1 contains all the accumulated properties from the proofs of all previous instructions. Applying H1 ends the first goal. The second element requires a value to be read of type *BE* by the *isBE* property. The check phase of the proofs gives this information. Notably, it looks for the block entry at the address where the value is to be read. If the block entry is found, it adds a type *BE* property and $blockAddr <> nullAddr$ property. If not, the address is considered to be default $nullAddr$. The accumulated properties then include the two possible paths as a disjunction:

```
1  blockAddr = nullAddr \/ (exists entry : BlockEntry, lookup blockAddr
   ↪  (memory s) beqAddr = Some (BE entry)
```

The two paths must be considered for the proof. For the first, $blockAddr = nullAddr$, obviously it cannot be true because a previous check has discarded that option. Indeed, the hypothesis H3 contains the information that $blockAddr <> nullAddr$. This path is then discriminated by contradiction.

The second path is trivially true after some definition unfolding (*unfold isBE*), hypothesis extraction (*destruct H12*) and rewriting (*rewrite H6*).

10.1.2 Proof of the security properties

Proof.

The same pattern is used for all instructions to propagate the accumulated properties and especially the security properties. Hence, we need not to consider the security properties individually because at the penultimate instruction they are still untouched from the initial state.

Every service exit is a *ret* instruction due to halts in the check phase or when reaching the last instruction. Indeed, recall that the Hoare triple holds, in other words we can terminate the proof, only in a nominal condition free of undefined behaviors.

```
1 { (** ret *)
2       eapply weaken. apply WP.ret.
3       intros. intuition.
4 }
```

Again, the proof script *weaken*s the properties and invokes the second-level proof of *ret*. Because the *ret* lemma in Listing 2 requires any properties to end, it terminates the proof.

Qed.

10.2 Proof of `addMemoryBlock`

The full proof is accessible online [3].

The proof goal is:

```
Lemma addMemoryBlock  (idPDchild idBlockToShare: paddr) (r w e : bool) :
{{fun s => consistency s /\ partitionsIsolation s  /\ kernelDataIsolation s
↪  /\ verticalSharing s }}
Services.addMemoryBlock idPDchild idBlockToShare r w e
{{fun _ s  => consistency s /\ partitionsIsolation s  /\
↪  kernelDataIsolation s /\ verticalSharing s }}.
```

The proof part concerning the security properties is accessible online [4].

[3] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/blob/addMemoryBlock_proof/proof/invariants/AddMemoryBlock.v

[4] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/blob/addMemoryBlock_proof/proof/invariants/AddMemoryBlockSecProps.v

10.2.1 Proof context

The state has deeply changed with kernel structure updates. Hence, the accumulated properties evolved after each state modification. However, the security properties depend on this modified state. Security properties are thus simply not in the proof context. In other words, even if the ultimate instruction is *ret* like in the findBlock service, the ending is not trivial anymore because the proof context does not know if the security properties still hold in the modified state.

Instead, the proof context contains all accumulated properties and state modifications from the initial state. Some properties emerge from the check phase similarly as described for findBlock. Other properties come from write primitives which modify the state. We study again the penultimate instruction involving the writeSh1InChildLocationFromBlockEntryAddr primitive as an example for the other modification instructions of the service.

```
writeSh1InChildLocationFromBlockEntryAddr blockToShareInCurrPartAddr
↪  blockToShareChildEntryAddr;;
```

The proof script of writeSh1InChildLocationFromBlockEntryAddr in Listing 12 is split in four parts:

- 1: Line 2: isolates the instruction from the rest of the service instructions
- 2: Line 5: *weaken*s the goal by invoking the second-level proof and invokes the latter. This ends the current goal because the second-level proof states that any properties can be propagated from its preconditions, which is kept abstract for the moment. The goal changes to match the second-level proof's preconditions. The new goal is then to prove these preconditions with the proof context depending on the new state *s*. Line 6: prepares the new proof context with state *s*.
- 3: Line 8: extracts information from the proof context that will be used for the proof. Proof that the proof context holds these properties to propagate and that they still hold in *s*. The state modification due to the operated instruction is propagated as well, as an update of the previous stored state. All state modifications are passed on until the last instruction.
- 4: Lines 10-20: proves all preconditions. Some properties to propagate depend on an ancient state which are trivially true. Some preconditions depends on invariants that must be locally proven.

Listing 12 Sketch of proof of `writeSh1InChildLocationFromBlockEntryAddr`

```
1  ( 1: isolate the instruction )
2  eapply bindRev.
3  { (** MAL.writeSh1InChildLocationFromBlockEntryAddr **)
4        ( 2: weaken the goal )
5        eapply weaken. apply writeSh1InChildLocationFromBlockEntryAddr.
6        intros. simpl.
7        ( 3: construct global knowledge )
8        (*...*)
9        ( 4: proof of the preconditions in the new state )
10       + ( Proof of: exists entry , lookup (CPaddr (blockentryaddr +
          ↪  sh1offset)) s.(memory) beqAddr = Some (SHE entry) )
11             (* ... *)
12       + ( select the properties to propagate and proof parametrised
          ↪  with the new state )
13          instantiate (1:= fun _ s => ...).
14             (*...*)
15       + ( Proof of: isBE blockentryaddr s )
16             (* ... *)
17       + ( Proof of: wellFormedFstShadowIfBlockEntry s )
18             (* ... *)
19       + ( Proof of: KernelStructureStartFromBlockEntryAddrIsKS s )
20             (* ... *)
```

The properties selected in the second part must be carefully chosen to propagate at least all the properties needed for the next instructions. For example, `writeSh1InChildLocationFromBlockEntryAddr` requires the *BlockEntryAddr*. The first and second-level proofs of the primitive indicate which invariants to use to get this property; however, the information about the block entry must be found in the proof context. Similarly as in `findBlock`, if the information that the block entry's presence was not delivered during the check phase or if this information was not propagated by previous instruction proof, then we would have missed a crucial property and we would not have been able to terminate the proof. The next chapter discusses how to reveal the essential properties to propagate.

10.2.2 Proof of the security properties

Proofs now reached the last instruction, *ret*, with a more furnished proof context than in the `findBlock` case.

The proof context contains accumulated properties from informational and modification primitives as well as all state modifications from the first instruction. Similarly as in `findBlock`'s proof, the *ret* instruction just passes on the accumulated properties.

```
{ (** ret **)
      eapply weaken. apply WP.ret.
      intros. simpl.
      ( 1: construct global knowledge for the isolation invariant )
      (* ... *)
      ( 2: prove the isolation invariant )
      -      ( 2.1: prove the consistency properties )
            (* ... *)
      -      ( 2.2: prove the security properties )
            (* ... *)
```

However, this time, the proof is not trivial because of the state modifications. The isolation invariant must be proven on the modified state *s* which is different from the initial state. The isolation invariant can then not be directly related to the initial state and the new proof goal is to prove the isolation invariant on *s*.

Proof goal:

```
consistency s /\ verticalSharing s /\ partitionsIsolation s /\
↪  kernelDataIsolation s
```

Assumptions: We assume the consistency properties hold to focus on the proof of the security properties. The curious reader finds the proofs of the consistency properties

in the full proof script. The proof of the security properties is split in three parts, one for each security property, as can be seen in the proof script.

There are common grounds for all security properties. A first phase is to extract from the proof context useful properties directly concerned by the security properties. This information notably recaps kernel element types at the modified addresses, which are then used to discriminate execution paths.

Vertical Sharing (VS)

Proof.

VS considers an arbitrary parent-child relationship. By introducing VS in the proof context (*intros* tactic), the definition of *incl* is unrolled so that the proof context also considers an arbitrary address *addr* within the child's *used* addresses.

```
1 HnAddrInUsedChild: In addr (getConfigPaddr child s ++ getMappedPaddr
  ↪  child s)
```

The proof goal becomes to demonstrate that *addr* also belongs to the parent's *mapped* addresses (union of all addresses in the mapped blocks only):

```
1 In addr (getMappedPaddr parent s)
```

The parent-child relationship is operated in `addMemoryBlock` to share a memory block with a child partition. In that case, both the parent and the child mutate. However, the child endures more modifications than the parent because all kernel elements of the child are touched by the modifications (Blocks, Shadow 1, Shadow Cut and Partition Descriptor) while only the Shadow 1 of the parent is updated to register the sharing. VS is interested in the inclusion of the child's mapped blocks and kernel structures in the parent's mapped blocks. The current partition and the designated child in `addMemoryBlock` can both take either the role of the parent or the child in VS, however, can not take the same role (because of the consistency property $NoDupInPartitionTree$). We understand the crucial modifications are actually the child's *Blocks* structure updates because it modifies the child's mapped blocks while the other kernel elements have no influence on that proof goal. In other words, the challenging parent-child combination in VS is the one where the child actually is the child in `addMemoryBlock`. We review in the following all possible combinations starting from the most difficult one.

child in VS is child in `addMemoryBlock` The proof starts by considering this combination:

```
1 destruct (beqAddr child globalIdPDChild) eqn:beqchildpd.
2 - (* child = globalIdPDChild *)
3     (* *)
```

This anchors the child in addMemoryBlock, globalIdPDChild, with the child in VS. We discard first the case where the parent in VS would also be globalIdPDChild leveraging the invariant *NoDupPartitionTree* 9.1.26 which forbids cycles in the partition tree.

```
assert(HparentidpdNotEq : parent <> globalIdPDChild). (* child not
  ↪ currentPart *)
```

The parent is then directly identified as the current partition.

```
assert(Hparent : parent = currentPart).
```

The current partition's mapped blocks have not changed during addMemoryBlock, so neither the union of their containing addresses.

```
assert(HmappedparentEq : getMappedPaddr currentPart s = getMappedPaddr
  ↪ currentPart s0).
```

This is not the case of the child globalIdPDChild which mapped blocks are augmented with a memory block, blockToShareInCurrPartAddr stemming from the parent (current) partition. Indeed, the global knowledge hypothesis states an address in the child's mapped blocks equivalently belongs either to the initial blocks or to the additional block (with start and end addresses respectively *(startAddr (blockrange bentry6))* and *(endAddr (blockrange bentry6))*:

```
assert(Hidpdchildmapped : forall addr,
                    In addr (getMappedPaddr globalIdPDChild s) <->
                    In addr
                    (getAllPaddrBlock (startAddr (blockrange bentry6))
                     ↪ (endAddr (blockrange bentry6))
                                ++ getMappedPaddr globalIdPDChild s0))
                    by intuition. (* constructed along the way *)
```

However, the child's configuration blocks have not changed, so neither their contained addresses:

```
assert(Hidpdchildconfigaddr : getConfigPaddr globalIdPDChild s =
  ↪ getConfigPaddr globalIdPDChild s0)
```

Thus, the hypothesis *HnAddrInUsedChild* can be rewritten as

```
HnAddrInUsedChild: In addr (getConfigPaddr globalIdPDChild s0 ++
  ↪ (getMappedPaddr globalIdPDChild s0 ++ getAllPaddrAux
  ↪ [blockToShareInCurrPartAddr] s))
```

with *getAllPaddrAux* collecting all addresses contained in a block. This can be done via hypothesis *HAddrInBTS*:

```
assert(HaddrInBTS :
   (forall addr : paddr,
  In addr
   (getAllPaddrBlock (startAddr (blockrange bentry6))
     (endAddr (blockrange bentry6))) <->
  In addr (getAllPaddrAux [blockToShareInCurrPartAddr] s0))) by
   ↪  intuition.
```

Now, we can discriminate the cases where the addresses belong to: 1) the initial configuration blocks, 2) the initial mapped blocks or 3) to the additional block `blockToShareInCurrPartAddr`:

```
apply in_app_or in HnAddrInUsedChild.
```

which are the concatenation of all addresses contained in the mapped blocks and in the kernel structures (*i.e.* ConfigBlocks) of that partition:

```
Definition getUsedPaddr (partition : paddr) s : list paddr :=
let ksList := getConfigPaddr partition s in
let mappedblockList := getMappedPaddr partition s in
ksList ++ mappedblockList.
```

The first and second cases are trivially true, assuming the security property was true at the initial state $s0$, with initial configuration blocks or mapped blocks.

```
assert(HVs0: verticalSharing s0) by intuition.
(* ... *)
specialize (HVs0 currentPart globalIdPDChild HparentPartTree
   ↪  HchildIsChild).
specialize (HVs0 addr).

(* ... *)

( Case 1: In addr (getConfigPaddr globalIdPDChild s0) )
apply HVs0. apply in_app_iff. left. assumption.

(* ... *)

( Case 2: In addr (getMappedPaddr globalIdPDChild s0) )
apply HVs0. apply in_app_iff. right. assumption.
```

To solve the last case, *i.e.* the address lies in the additional block `blockToShareInCurrPartAddr`, we must first state that the address lies in the parent's initial block

because addMemoryBlock picked up all the additional block's characteristics from the block in the parent (the service clones the block in the child):

```
assert(HparentInMappedlist : In blockToShareInCurrPartAddr
  ↪ (getMappedBlocks currentPart s0)) by intuition. (* found block in
  ↪ the parent *)
```

Induction on the mapped blocks of the parent solves the last proof goal:

```
induction (getMappedBlocks currentPart s0).
(* ... *)
```

child in VS is the parent in addMemoryBlock Another possible combination is to get one level up in the Pip partitioning scheme and consider the parent in VS to be the child partition globalIdPDChild in addMemoryBlock (and so the current partition is the grand-parent of child in VS, so outside the scope of the proof goal).

```
- (* child <> globalIdPDChild *)
      destruct (beqAddr parent globalIdPDChild) eqn:beqparentpd.
      -- (* parent = globalIdPDChild *)
```

In other words, the child in VS could be any partition not addressed in addMemoryBlock.

Again, the possible identification of globalIdPDChild and this time the child in VS is eliminated via the *NoDupPartitionTree* invariant.

```
assert(HNoDupPartTree : noDupPartitionTree s) by (unfold consistency
  ↪ in * ; unfold consistency1 in * ; intuition).
assert(HglobalChildNotEq : globalIdPDChild <> child).
```

This leads to the properties stating that the child in VS is untouched by addMemoryBlock operated from the grand-parent (current) partition, and so the address lies in the initial mapped blocks of the parent globalIdPDChild.

```
assert(HusedchildEq : getUsedPaddr child s = getUsedPaddr child s0).
(* ... *)
assert(HVs0: verticalSharing s0) by intuition.
(* ... *)
specialize (HVs0 globalIdPDChild child HparentPartTree HchildIsChild
  ↪ addr HnAddrInUsedChild).
(* HVs0: In addr (getMappedPaddr globalIdPDChild s0)*)
```

However, we know from the global knowledge properties that the mapped addresses from the parent globalIdPDChild can be rewritten as the concatenation of initial mapped addresses with the additional block *blockToShareInCurrPartAddr* as seen above. The proof follows trivially by rewriting the proof goal.

```
assert(Hidpdchildmapped : forall addr,
                    In addr (getMappedPaddr globalIdPDChild s) <->
                    In addr
                    (getAllPaddrBlock (startAddr (blockrange bentry6))
                     ↪  (endAddr (blockrange bentry6))
                              ++ getMappedPaddr globalIdPDChild s0))
                    by intuition. (* constructed along the way *)
specialize (Hidpdchildmapped addr).
rewrite Hidpdchildmapped.
(* using HVs0: In addr (getMappedPaddr globalIdPDChild s0)*)
apply in_or_app. (* Proof goal: In addr
                          (getAllPaddrBlock (startAddr (blockrange
 ↪  bentry6)) (endAddr (blockrange bentry6)) \/ In addr
 ↪  (getMappedPaddr globalIdPDChild s0)*)
right. (* select second option *)
assumption.
```

Any other combination not affected by `addMemoryBlock` The remaining combinations do not associate the child or parent in VS with any partitions manipulated by `addMemoryBlock`.

```
-- (* parent <> globalIdPDChild *)
   (* ... *)
```

This last case is trivially solved because all elements are not touched by `addMemoryBlock` and so can be related to the initial state where it is assumed to be true.

```
assert(HVs0: verticalSharing s0) by intuition.
(* ... *)
assert(HchildrenparentEq : getChildren parent s = getChildren parent
 ↪  s0).
(* ... *)
assert(HusedchildEq : getUsedPaddr child s = getUsedPaddr child s0).
(* ... *)
assert(HmappedparentEq : getMappedPaddr parent s = getMappedPaddr
 ↪  parent s0) by (apply HMappedPaddrEqNotInParts0 ; intuition).
(* ... *)
rewrite HusedchildEq in *.
rewrite HmappedparentEq in *.
rewrite HchildrenparentEq in*.
```

```
12  specialize (HVs0 parent child HparentPartTree HchildIsChild addr
    ↪  HnAddrInUsedChild).
13  assumption.
```

Qed.

Horizontal Isolation (*HI*) (also called *partitionsIsolation* by inheritance)

Proof.

HI introduces a supplementary parent-child relationship compared to *VS* which increases the number of combinations. The first part of the proof script extracts useful information from the proof context that will help discard impossible combinations due to wrong types or not respecting the *NoDupPartitionTree* invariant to not have any cycles in the partition tree.

Introducing arbitrary values for *parent*, *child*1 and *child*2 gives the following proof goal:

```
1  disjoint (getUsedPaddr child1 s) (getUsedPaddr child2 s).
```

Again, the proof script glides the parent-child relationship of addMemoryBlock (current partition and globalIdPDChild) on *HI*'s parent-children relationships.

*child*1 **in** *HI* **is the child in** addMemoryBlock **and** *child*2 **is anything**

```
1  destruct (beqAddr child1 globalIdPDChild) eqn:beqchild1pd.
2     - (* child1 = globalIdPDChild *)
```

In this configuration, the *parent* in *HI* is the current partition of addMemoryBlock.

```
1  assert(Hparent : parent = currentPart).
2  { (* ... *) }
```

Because addMemoryBlock just considers one parent-child relationship, all other partitions are untouched. This is the case for the addresses contained in their mapped blocks as resumed by hypothesis *HUsedPaddrEqNotInParts0*:

```
1  forall partition : paddr,
2        partition <> globalIdPDChild ->
3        isPDT partition s0 ->
4        getUsedPaddr partition s = getUsedPaddr partition s0
```

Thus the proof goal can be rewritten as *In addr (getUsedPaddr child2 s0)* by further unfolding of *Lib.disjoint*:

```
assert(Husedchild2Eq : getUsedPaddr child2 s = getUsedPaddr child2 s0).
{ apply HUsedPaddrEqNotInParts0 ; intuition. }
(* ... *)
unfold Lib.disjoint.
intros addr HaddrInchildused.
(* ... *)
rewrite Husedchild2Eq in *.
```

Currently, the proof context has the information that *addr* lies in the *used* addresses of `globalIdPDChild` in the modified state (hypothesis *HaddrInchildused*) and we must prove *addr* is then not in *partition2*'s *used* addresses at initial state *s*0. We rewrite *HaddrInchildused* similarly as in the proof of *VS* and differentiate the cases when *addr* lies in 1) the initial configuration blocks, 2) the initial mapped blocks or 3) the new added block `blockToShareInCurrPartAddr`:

```
apply in_app_or in HaddrInchildused.
destruct HaddrInchildused.
```

The first two cases follows immediately after a specialisation of *HI* at *s*0.

```
assert(Hpartisolations0: partitionsIsolation s0) by intuition.
(* ... *)
(* Case 1: In addr (getConfigPaddr globalIdPDChild s0) *)
unfold Lib.disjoint in Hpartisolations0.
specialize (Hpartisolations0 addr).
apply Hpartisolations0.
unfold getUsedPaddr. intuition.
(* ... *)
(* Case 2: In addr (getMappedPaddr globalIdPDChild s0)*)
unfold Lib.disjoint in Hpartisolations0.
specialize (Hpartisolations0 addr).
apply Hpartisolations0.
unfold getUsedPaddr. intuition.
```

The last case requires additional preparation of the proof context. Indeed, *addr* lying in the new block of the child `globalIdPDChild` at modified state implies *addr* was also contained in the parent (current) partition at *s*0 because `addMemoryBlock` clones the block in the parent from *s*0 into the child. In other terms, we must prove `blockToShareInCurrPartAddr` was not already shared with *child2* at *s*0. We prove this by contradiction so we assume that *addr* lies in *child2*'s *used* addresses at initial state *s*0: *HaddrInChild2s0: In addr (getUsedPaddr child2 s0)*.

The block is known to belong to the mapped blocks of the parent at s0:

```
assert(HparentInMappedlist : In blockToShareInCurrPartAddr
  ↪ (getMappedBlocks currentPart s0)) by (rewrite HcurrPartEq in * ;
  ↪ intuition). (* by found block *)
```

In the proof context, the block is then in the parent's and the *child2*'s mapped blocks, so shared with *child2* at *s0*. Invariant *sharedBlockPointsToChild* implies the *Shadow 1* entry in the parent to refer to the child, either by the *PDflag* or the *PDchild* fields. However, the check phase revealed the block was not shared at *s0* which concludes the proof with a contradiction in the proof context.

```
assert(HsharedInChilds0 : sharedBlockPointsToChild s0) by (unfold
  ↪ consistency in * ; unfold consistency1 in * ; intuition).
(* ... *)
assert(Hsh1entryaddr : sh1entryAddr blockToShareInCurrPartAddr
  ↪ sh1eaddr s0).
{ (* ... *) }
specialize (HsharedInChilds0 currentPart child1 addr
  ↪ blockToShareInCurrPartAddr sh1eaddr HparentPartTree
  ↪ Hchild1IsChild HaddrInchildused HaddrInParentBlock
  ↪ HparentInMappedlist Hsh1entryaddr).

destruct HsharedInChilds0 as [Hsh1entryaddrs0 | Hsh1entrychilds0].
(* ... *)
+ (* case pdchild = child1 in the block's **Shadow1** entry*)
    (* contradiction because check phase stated that pdchild =
    ↪ nullAddr *)
+ (* case pdflag = true in the block's **Shadow1** entry  *)
    (* contradiction because the check phase stated that pdflag =
    ↪ false *)
```

child1 **in** *HI* **is anything and** *child2* **is the child in** addMemoryBlock

```
- (* child1 <> globalIdPDChild *)
      destruct (beqAddr child2 globalIdPDChild) eqn:beqchild2pd.
      -- (* child2 = globalIdPDChild *)
```

Like observed during the informal proof, this is the symmetric case that we proved just before.

*child*1 **and** *child*2 **in** *HI* **are any partition**

```
    (* ... *)
    -- (* child2 <> globalIdPDchild *)
```

This combination considers any children of *parent* that are not `globalIdPDChild`. In the contrary, *parent* has no restriction, so could be `globalIdPDChild` or any other partition.

parent **in** *HI* **is child in** `addMemoryBlock`

```
destruct (beqAddr globalIdPDChild parent) eqn:beqparentidpd.
--- (* globalIdPDChild = parent *)
```

Potential children of `globalIdPDChild` are untouched by the operation of `addMemoryBlock`.

```
assert(Husedchild1Eq : getUsedPaddr child1 s = getUsedPaddr child1 s0).
{ apply HUsedPaddrEqNotInParts0 ; intuition. }
assert(Husedchild2Eq : getUsedPaddr child2 s = getUsedPaddr child2 s0).
{ apply HUsedPaddrEqNotInParts0 ; intuition. }
```

The proof goal can be rewritten to the property at $s0$ when the property was assumed true. A specialisation of the property with the arbitrary introduced values terminates the proof.

```
specialize (Hpartisolations0 globalIdPDChild child1 child2
  ↪ HparentPartTree Hchild1IsChild Hchild2IsChild
  ↪ Hchild1child2NotEq).
rewrite Husedchild1Eq.
rewrite Husedchild2Eq.
assumption.
```

parent **in** *HI* **is any partition**

```
--- (* globalIdPDChild <> parent *)
```

This is the global case with only untouched partitions. Likewise to the previous proof, we can conclude by rewriting the proof goal to the equivalent $s0$ property and by specialising the security property at $s0$.

Qed.

Kernel Data Isolation (*KI*)

Proof.

The informal proof led us to relate the security property on the modified state *s* to the initial state *s*0 because no kernel structure are added or removed in addMemoryBlock.

The expression of the security property is different than the other security properties in that not only parent-child relationships are considered in *KI* but also relationships with itself. Thus, the cases that were discarded in the previous proofs because of the parent-child relationship restriction must be investigated in this new situation.

After introduction of all hypotheses, the goal is to prove, for any partition combination, that any address *addr* contained in the mapped blocks of the first partition do not lie in a configuration block of the second partition:

```
~In addr (getConfigPaddr partition2 s)
```

knowing the hypothesis *HaccessiblePaddr: In addr (getAccessibleMappedPaddr partition1 s)*.

*partition*1 **and** *partition*2 **in** *KI* **is child in** addMemoryBlock The proof starts with the following combination:

```
destruct (beqAddr part1 globalIdPDChild) eqn:beqpart1pd.
      - (* part1 = globalIdPDChild *)
         (* ... *)
         destruct (beqAddr part2 globalIdPDChild) eqn:beqpart2pd.
                  -- (* part2 = globalIdPDChild *)
```

Within the same partition, *addr* must be absent from configuration blocks. Furthermore, we know the configuration blocks have not changed for any partition, so in particular not for globalIdPDChild:

```
assert(Hidpdchildconfigaddr : getConfigPaddr globalIdPDChild s =
 ↪  getConfigPaddr globalIdPDChild s0)
                  by intuition. (* constructed along the way *)
(* ...*)
rewrite Hidpdchildconfigaddr in *.
```

The new proof goal is then:

```
~In addr (getConfigPaddr globalIdPDChild s0)
```

This shows the intuition was correct since we now have to demonstrate a property on the initial state *s*0.

We leverage the freedom to choose any partition combination at $s0$ to engage the current partition:

```
assert(HKIs0: kernelDataIsolation s0) by intuition.
(* ... *)
specialize (HKIs0 globalIdPDChild globalIdPDChild Hpart1PartTree
  ↪ Hpart2PartTree).
specialize (HVs0 addr).
```

The proof context now contains $HKIs0$: $In\ addr\ (getAccessibleMappedPaddr\ currentPart\ s0) \implies In\ addr\ (getConfigPaddr\ globalIdPDChild\ s0)$. This would end the goal if the first statement is proven true. In other words, *addr* which is in the accessible memory blocks of `globalIdPDChild` of the modified state (hypothesis *HaccessiblePaddr*) must also be in the accessible memory blocks of the current partition at the initial state.

To prove that, we extract the information that `globalIdPDChild` accessible addresses can be rewritten as a composition of initial addresses with addresses from the newly added block `blockToShareInCurrPartAddr`:

```
assert(HMappedPaddrEq : In addr (getAccessibleMappedPaddr
  ↪ globalIdPDChild s) ->
In addr ((getAllPaddrBlock (startAddr (blockrange bentry6)) (endAddr
  ↪ (blockrange bentry6)))
++ (getAccessibleMappedPaddr globalIdPDChild s0)))
      by intuition.
specialize (HMappedPaddrEq HaccessiblePaddr).
```

We can then explore the independent cases where the address 1) lies in the initial mapped blocks or 2) in the new block.

```
apply in_app_or in HMappedPaddrEq.

destruct HMappedPaddrEq as [HaddrInNewB | HaddrInMappedPaddrs0].
```

In both cases, we can prove the missing hypothesis:

```
assert(HaddrInAccessibleParent : In addr (getAccessibleMappedPaddr
  ↪ currentPart s0)).
```

Indeed, case 1 is trivially true via the *accessibleChildPaddrIsAccessibleIntoParent* invariant 9.1.29 because any accessible address in the child `globalIdPDChild` is also accessible in the parent (current) partition:

```
assert(HaccessibleInParents0 :
  ↪ accessibleChildPaddrIsAccessibleIntoParent s0) by (unfold
  ↪ consistency in * ; unfold consistency1 in * ; intuition). (*
  ↪ consistency s0*)
eapply HaccessibleInParents0 with globalIdPDChild ; intuition.
```

Case 2 is also trivially true because the new block stems from the parent (current) partition, and we know from the check phase the block was accessible at $s0$. Hence, all addresses contained in this block, especially *addr*, are part of the accessible mapped blocks (lemma *addrInAccessibleBlockIsAccessibleMapped*).

```
(* extract accessible information *)
assert(addrIsAccessible = true) by (apply negb_false_iff in
  ↪ Haccessible ; intuition).
(* ... *)
apply addrInAccessibleBlockIsAccessibleMapped
                  with blockToShareInCurrPartAddr ; intuition.
```

$partition1$ in KI is child in addMemoryBlock and partition 2 is anything

```
(* still in case part1 = globalIdPDChild *)
   -- (* part2 <> globalIdPDChild *)
```

Similarly, for any other partition, the configuration blocks have not changed so we can leverage the hypothesis $HConfigPaddrEqNotInParts0$:

```
forall partition : paddr,
      partition <> globalIdPDChild ->
      isPDT partition s0 ->
      getConfigPaddr partition s = getConfigPaddr partition s0.
```

The proof goal can be rewritten as *In addr (getConfigPaddr partition2 s0)* by unfolding the definition of *Lib.Disjoint*:

```
assert(Hidpart2configaddr : getConfigPaddr part2 s = getConfigPaddr
  ↪ part2 s0)      by (eapply HConfigPaddrEqNotInParts0 ; intuition).
(* ... *)
rewrite Hidpart2configaddr in *.
unfold Lib.disjoint in *.
intros addr HaccessiblePaddr.
```

Exactly in the same way as the previous combination, we use the assumption that the security property was true at initial state $s0$ and specialize it with the current partition as $partition1$ to retrieve the proof goal in the proof context (because we rewrote the goal to prove that the property was true at $s0$, see above):

```
assumption.
```

*partition*1 **in** *KI* **is anything while** *partition*2 **is child in** `addMemoryBlock`

```
- (* part1 <> globalIdPDChild *)
    (* ... *)
    destruct (beqAddr part2 globalIdPDChild) eqn:beqpart2pd.
       -- (* part2 = globalIdPDChild *)
```

In this combination, the proof is immediate because 's configuration blocks are unchange, so the same as in the initial state at $s0$ when the property is assumed true.

```
assert(HKIs0: kernelDataIsolation s0) by intuition.
(* ... *)
assert(Hidpdchildconfigaddr : getConfigPaddr globalIdPDChild s =
  ↪ getConfigPaddr globalIdPDChild s0)
                    by intuition. (* constructed along the way *)
(* ... *)
rewrite Hidpdchildconfigaddr in *.
specialize (HKIs0 part1 globalIdPDChild Hpart1PartTree
  ↪ Hpart2PartTree).
intuition.
```

*partition*1 **in** *KI* **is anything and** *partition*2 **in** `addMemoryBlock` **is anything**

```
- (* part1 <> globalIdPDChild *)
    (* ... *)
    -- (* part2 <> globalIdPDChild *)
```

Exactly as previous proof, for any other partition, the configuration blocks are unchanged and so the proof goal is equivalent to proving the property at the initial state $s0$ assumed true:

```
assert(Hconfig2Eq : getConfigPaddr part2 s = getConfigPaddr part2 s0)
  ↪ by (eapply HConfigPaddrEqNotInParts0 ; intuition).
rewrite Hconfig2Eq in *.
specialize (HKIs0 part1 part2 Hpart1PartTree Hpart2PartTree).
intuition.
```

Qed.

10.3 Evaluation of Pip-MPU's proof development

We evaluate the proof development with common metrics linked to formal verification.

10.3.1 Proof setup

The proofs are developed using the CoqIDE 8.13.1 on an ArchLinux 5.18.1 distribution. The tools run on an HP ZenBook G4 17 provided with a 64-bit Intel i7-7820HQ 4-core CPU, 16 GB RAM, 16 GB swap space, 155 GB SSD usable space. There is one proof script per service and per internal function (inner functions called by the services). There is one proof script for each proof level of the HAL primitives. There is one additional proof script per modification service to prove the security properties apart from the consistency properties.

10.3.2 Results

Two services have been completely proven, `readMPU`, `findBlock`, and we have proved all security properties and most of the consistency properties in `addMemoryBlock`.

Table 10.1 provides proof metrics on the proof development process (formalisation and the verification stages) and summarizes the current status of the proof development.

The formal verification of Pip-MPU is an ongoing work. At the time of writing, two services out of the fifteen implemented services have been fully proven, and an additional one is close to completion. The metrics on the invariants provide information on the number of consistency properties and security properties. The invariants are also written (specified) in Gallina and the number of SLOC reports the total size of the properties, trimmed from comments.

The formalisation stage considers three phases: the HW specification understanding, the design phase, and the Coq model implementation. The first phase consisted in collecting, reading and combining different sources on the target HW platform [19, 14, 176, 130, 131]. The design phase includes any conversations, thinking time, methodological reflections, paper trials that finally converged to services in pseudo-code which were eventually translated in Python scripts for simulation purposes. The last phase required a full model implementation by forking the Pip (MMU) project [144] and by adapting the model to a hardware without virtual memory and modifying the kernel structures to match Pip-MPU's design, given the Python implementation. Duration is estimated based on status advancement reports and code versioning history.

The proof development phase focuses on the three services that underwent a complete formal verification. We do include in our time estimation the time spent to prove the proof levels of the low level HAL primitives because they are proved along the way,

Proof status				
Services				
# services	15			
# fully verified services	2			
Invariants				
# consistency properties	27			
# security properties	3			
Total invariant properties	29			
Consistency properties SLOC	129			
Security properties SLOC	14			
Total properties SLOC	143			
Proof development				
Formalisation				
Duration of HW specification understanding	2 months			
Duration of design phase	9 months (including 2 months Python simulation)			
Duration of model implementation in Coq	2 months			
Duration of service development in Coq	2 months			
	readMPU	**findBlock**	**addMemoryBlock**	
# SLOC (w/o HAL)	11	14	25	
Proofs	*All properties*		*Consistency properties*	*Security properties*
# LOP (Lines of Proof)	76	81	6143	1516
# tactics	90	93	6838	759
Ratio #LOP/SLOC	6.9	5.8	246	60.4
Ratio #tactics/SLOC	8.18	6.6	273	30.4
Duration of proof development	2 days	4 hours	7 months (ongoing)	4 days
person-month (estimate)	0.09	0.02	5.5	0.18
Ratio person-month/LOC	0.008	0.0014	0.22	0.0072
Duration of proof compilation	0.5s	0.4s	939s (15.65 min)	127s (2.1 min)
Memory footprint during proof compilation	400 MB	400 MB	5000 MB	900 MB
CPU usage during proof compilation	100%	110%	115 %	105%
Proof coverage (estimate)	100%	100%	90%	100%

Table 10.1: Proof status and effort.

when used by a service. However, because the primitives are very similar, the individual proofs are basically small adaptations of the proofs on other primitives, so the duration is negligible compared to the development of the services. The proof development metrics gather the number of Gallina lines of the services and corresponding Coq proof lines to prove them (without internal functions and lemmas). The number of lines is computed after removing all comments. However, this method does not really capture the number of necessary steps in the proof development process because a great number of variables are split over several lines and because there could be several tactics on the same line. So we also estimate the proof effort regarding the number of used *tactics*. The *tactics*, also called hints, are terms of the language used by the prover and the way the proof developer interacts with it. They are instructions capturing the logical reasoning and intuition of the proof developer in understandable language to the prover. The number of tactics is computed by removing all comments and proof specifications and counting the number of instructions ending with a dot or semi-colon. We provide the ratio between the service's source code lines and the number of Lines of Proof (LoP) as well as with the hints to give a sense of the proof script size compared to the service code base, again without considering inner elements (functions or corresponding lemmas).

We report the human time spent during the verification of each service estimated via the code versioning history. Note that *findBlock* has about the same number of SLOC and LOP than *readMPU*, but is proved in some hours instead of days. Indeed, they share a lot of common functions and lemmas, and so the *findBlock* proof recycled a lot from the previous proof, drastically diminishing the proof effort.

The compilation of all proofs on the host machine in charge of the proof development is expressed in time, CPU usage and memory footprint. The measurements are recorded by using the `psrecord` tool[5], as illustrated with `addMemoryBlock` in Figure 10.1. They do not include the compilation time to prove their dependency lemmas (that could be common between different services), just the main lemma.

The proof coverage estimation gives the completion rates of each service. About `addMemoryBlock`, the current status at time of writing is that all security properties have been verified on the modified state assuming all consistency properties hold. The latter are currently in the final stage of verification.

[5] `https://github.com/astrofrog/psrecord`

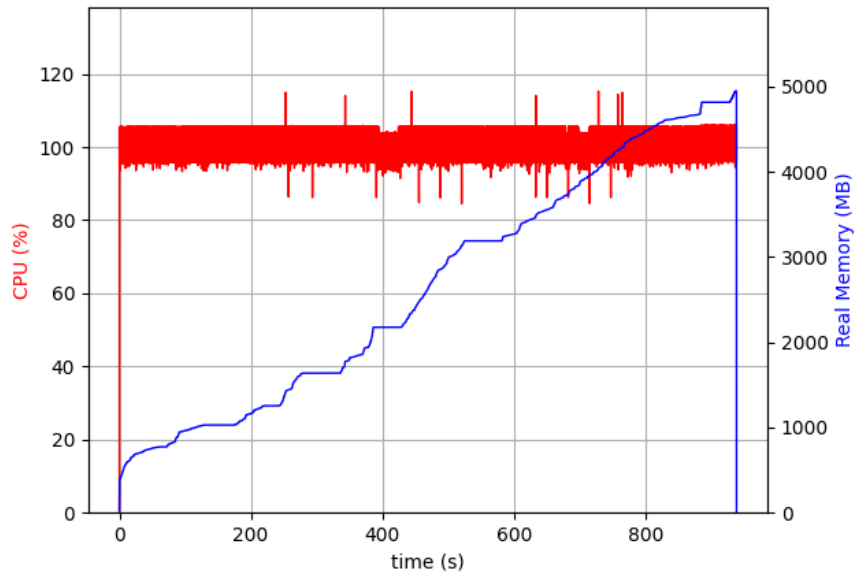


Figure 10.1: CPU and memory usage during the proof compilation of `addMemoryBlock`.

10.4 Discussion and limitations

10.4.1 Proof status

The proof status reports the current status. The consistency properties are the minimal set of properties to prove the security properties for the services. However, for reasons discussed in the next chapter, they are believed to be incomplete even though close to the complete set of properties. The maintenance needed to include new properties or modify the current ones has a direct effect on all proof metrics.

10.4.2 Proof development

Many parts of the proof script were eluded for the sake of clarity by focusing on the crucial moments of the proof. Additional constraints on kernel structures, for example bounded values for a certain type, are not discussed here but are present in the full proof script.

Furthermore, the proof script reflects the proof goal we want to achieve, namely the security properties. However, how can we be sure that the expression of the properties really corresponds to the intended properties? Indeed, we tend to simplify the model to have simplified abstract reasoning capturing the complexity of our thoughts. There is currently no way to be sure about that and we can prove incorrect specifications,

even false ones. Nevertheless, the limited set of security properties and their simplicity convince us of the correctness of the proofs.

Moreover, only two services have been completely proven whereas the other ones are partially verified (one close to completion). However, they lay the foundations for the other services. Indeed, all purely informational services have been proven while `addMemoryBlock` represents modification services, of the type that creates kernel elements. A future challenge is to show the proof framework is also correctly working for the type of service that destructs kernel elements, such as `removeMemoryBlock`.

10.4.3 Proof metrics

Presented proof metrics give an idea of the underlying proof effort, however, must not be taken as raw metrics without further discussion.

Concerning the Coq model implementation, the HAL primitives are introduced as the proof of the services evolve, *i.e.* proof scripts are developed when needed and not by anticipation. This methodology is discussed in the next Chapter 11 about proof techniques. Meanwhile, it means the formalisation is still ongoing and the duration reflects the time spent to cover the three services fully addressed until now.

Concerning the number of lines in the proof script, while it definitely shows a quantitative measure of the proof effort, it does not indicate a transferable metric on other systems or other developers that would develop the proofs differently according to their style (for example in terms of structure, proof context reorganisation, or use of lemmas to generalise situations). Duplicates are clearly indicated in the proof script for future maintenance. The goal of this work was to prove the security properties, however the means to achieve it are infinite and not optimised. In other words, many parts are copies of other proofs, wordy, and might only exist for human clarity while being useless for the prover, which does not reflect neither the time spent on the proof nor the difficulty of the task. This is also emphasised in other formal verification projects [29].

The number of tactics is closely related to the previous remark. It also has the additional effect that some tactics are packed together by semicolons to solve induced sub-cases at the same time. The remark also affects the ratio of SLOC over the number of tactics.

The proof development was not a sequential process. The duration metric is an addition of all the times the proofs were developed. Moreover, the duration is impacted by the proof framework set-up (enrichment in internal lemmas), which becomes more and more effective due to permanent refactoring and optimisation discussed in the next chapter. Thus, the proof duration is included in the absolute proof development time, but the person-month metric estimates the actual time spent on the respective proofs. The person-month/SLOC ratio gives us an idea of the average effort to prove a

single line of code. In Pip-MPU, we can take the overall person-month estimate (5.79) and the current number of verified instructions (50) to compute the estimate of 0.12 person-month by line of code. We can compare with the effort required in other formally verified kernels, such as seL4 and CertiKOS [107, 85] In seL4, they report roughly 20.5 person-years to prove the functional correctness of a base code of about 10000 lines of code, of which around 7500 were verified. Then, about 4.1 person-years were needed for the security proofs. All in all, it gives the estimate of around 0.04 person-month per line of verified code. In CertiKOS, they report 2 person-years for 6500 lines of C and assembler for the mC2 kernel, which gives a ratio of 0.0037 person-month per line of code. While the comparison seems bad for Pip-MPU, it should not be taken as a reliable estimate. Indeed, the comparison is between two projects that ended their verification and gathered experience, reusable elements and optimisations along the way that are not shown but included in the metrics. Furthermore, the reported time spent on the project does not include helper tools, wherein automation is made possible, that accelerate proof development. Pip-MPU only uses the raw functionalities of Coq. In addition to that, we could compare instead the time spent just to prove the security properties, as it is done separately in seL4 on the abstract model and directly on the concrete model in Pip-MPU. For seL4, this leads to a ratio of 0.0066 person-month per line of code, whereas in Pip-MPU it is at worst 0.0072 for `addMemoryBlock`, so very close, and because there is no proof effort in `readMPU` and `findBlock` it leads to an average of 0.0036 for the time being, so twice better.

The proof compilation phase takes lots of resources, especially computing power and RAM. CoqIDE crashed many times due to insufficient resources or froze the machine because of heavy computations. Already from the previous Pip (MMU) project, proof scripts were tailored to fit 32 GB RAM. We followed the same approach in Pip-MPU by splitting proofs over different files: one file per service or function, and the proofs of the security properties alone. More than plain hardware requirements, this proof breakdown allowed to distribute the work and work in parallel on different proof scripts. This work organisation enables different proof developers to work on different parts of the proof at the same time and progress differently on scripts without having to comment code or using mock-ups.

10.4.4 Proof assumptions

Implicitly mentioned in previous sections and chapter, the proof relies on a lot of assumptions that we explicitly exhibit in this section. Incorrect assumptions imply a false context, which means everything can be proven out of it. Our proof goal for Pip-MPU thus depends on correct assumptions.

Assumptions are mostly similar to Pip (MMU version) [99] and other formally

verified kernels [89, 41, 114] in a broader perspective.

Indeed, the trustworthiness of the proofs depends on:

- the used tools
- the correctness of the model and especially the correspondence between the hardware model and the hardware platform as well as between the Coq HAL and its C/ASM implementation
- the hardware implementation and deltas with its specification
- the specification of the isolation invariant
- the atomicity of the services
- the bootstrapping phase.

First, tools must be trusted, let it be the Coq kernel, Digger, gcc or even the IDEs (Integrated Development Environments) we are using. Digger provides a literal translation from Gallina to C which we can check thanks to the minimal size of the kernel code base. One could increase the level of trust by replacing gcc by CompCert (a formally proven C compiler) and by replacing Digger by δx (a formally proven translator from C to CompCert's Abstract Syntax Tree), which are ongoing works. Critical bugs could be discovered in the Coq kernel (as discovered in the past), however the Coq community is obviously very concerned about these threats.

Second, we considered a simplified hardware model (memory layout, MPU) which captures the basic functionalities of the hardware platform. There are two major considerations for this item. The first consideration is that this simplification might be too simplistic and put aside effects that might endanger our proofs. One way to increase confidence would be to play the proofs on a formal model of the hardware. It could be directly handed over by processor manufacturers, like recent efforts on the ARMv8-A architecture [147], on which we would directly "plug" our proofs. However, we trust our hardware model in that it represents simple objects that facilitate the proofs and is a common representation, even between hardware vendors. Our hardware state assumes the kernel manipulates a writable non-volatile memory. In the opposite case (kernel structures in Read-Only Memory or losing information during the execution), the check phase would not be reliable and so would the proofs. Furthermore, we assume no external influence can affect memory without bypassing the MPU control. In particular, we consider either peripherals like the Direct Memory Access (DMA) not existing or disabled, like other verified kernels [29, 162]. Recent research [25, 88] showed the DMA could be used even in strict secure environment. Pip (MMU) already does this by controlling that the configuration registers of the DMA are valid to the partition accessing

them and is currently being adapted to Pip-MPU by our development team. The second consideration is that the low-level primitives of the HAL might not reflect the real C implementation that is manually written and so not checked. In other words, wrongly implemented C primitives could do something else than what was expected in the Coq model. The probability of such an event is considered very low because the primitives are composed of a small amount of C lines (around 5 lines on average) and directly correspond, literally, to their Coq model counterparts, like `readPDTable` in C and in Coq. Moreover, kernel structures modeled in Coq differ from kernel structures written in C. We believe their representation in both worlds to be accurate given constraints imposed on the Coq kernel structures to reflect C types. We specified in Coq that the address 0 has the *PADDR* type to represent the NULL pointer. No kernel structures can then be stored at that address, which is the case in the ARM implementation with 0 being reserved for the reset instruction.

Third, even if the proofs are conducted on a trusted model, we must trust a correct implementation of the hardware platform. Not formally verified hardware specification could contradict itself, or the silicon vendor could not implement exactly what the specification requires (diligently because many possibilities are *implementation defined* or by negligence) or could even introduce backdoors, which would all endanger the proof base. Furthermore, the hardware might not work as expected even by strictly following formal models because of physical failures or attacks, but this is out of scope for this work. We assume that the MPU is working correctly and that the memory controller correctly writes and reads the values that are passed by the HAL primitives. This holds for the target platform but also for the development platform.

Fourth, the isolation invariant, with the security properties and all consistency properties, could be incorrect. However, many similarities exist between Pip (MMU) and Pip-MPU, while being developed independently and challenged to achieve the same proof goal, which strengthens the confidence in both proof specifications. Pip (MMU) proved an equivalence with the isolation formalised by Rushby [155, 154] and so we are confident about our specification. We could also argue about the limited number of invariants and their small size that convince us they are correct. In the end, the most problematic issue with the specification is a false interpretation. From the formal definitions, to human language to express it and popularise the concept, until the reception and interpretation of the message by the interested reader or listener, there might be a comprehension gap, undermining the proofs and what they really demonstrate.

Finally, we assume a secure state at each service execution, which implies the bootstrapping routine to prepare the system into this secure state. This involves a correct preparation and launch of the root partition which satisfies the isolation invariant. In

particular, no Pip-MPU memory blocks are given to the root partition. The code base is limited (see Table 7.3) and we take extra care to bootstrap the system which should lower the risk of an assumption deviation.

To summarize, the developed proofs rely on common assumptions for formally verified kernels. Some of them are mitigated either by replacing tools with more trustworthy ones or by setting up procedures to control the code base. The remaining identified assumptions are outside the scope of this work or outside our control, but we acknowledge efforts to reduce the gap between assumptions and reality which are, moreover, closely monitored by the other verified kernels.

10.4.5 Bug discovery during the verification

The verification process unveiled a bug in the `addMemoryBlock` service. Indeed, the check phase omitted to verify if the block to be shared with the child partition was not already shared with another partition. This would have had the undesired state that a block is present in several children which breaks the isolation property. As a matter of fact, the bug was discovered during the proofs of the security properties, where it was impossible to conclude. This demonstrates the very essence of formal verification to detect such situations. It also exhibits that severe bugs can be found even in a small code base which focuses on security. Bug discovery is documented in other formal verification projects [99, 29, 109].

This bug also stresses what is and what is not claimed in this formal verification process. The only properties that are proved are the isolation invariant and no proofs concern functional properties. This remark also refers to other security policies than Pip's, for example the isolation invariant is not concerned about the read, write, and execution rights of the memory blocks.

10.5 Conclusion

This chapter focused on the proof of the security properties using the Coq Proof Assistant.

Two selected services illustrate the formal reasoning that closely follows the informal reasoning of the previous chapter. They are examples framing the proof development of both purely informational services and modification services. While the transcribed proof script renders important moments of the proof, many details have been omitted for the sake of clarification. However, thanks to the mechanized proof framework, the full proofs are available publicly and externally checked using Coq. Proofs are done at implementation level.

Proofs deeply question the assurance of our design. Indeed, by exhibiting the assumptions in which the proofs are rooted, we must be confident with a small gap with reality. However, we showed and are convinced the set of assumptions is limited, can be further minimised, and the research community invests in the remaining identified concerns. We created a simple formal model for the MPU because its formal specification does not exist yet and we join Dijkstra who once said: "to whom of the two [physical equipment and formal system] we give the primacy, that is whether it is the task of the formal system to give an accurate description of (certain aspects of) the physical equipment, or whether it is the task of the physical equipment to provide an accurate model for the formal system —and I prefer the latter—" [60].

With a small TCB, the proof efforts show manageable proof development for the kernel without the need for refinement techniques. Proof efforts were considerably larger than the development of the services themselves. However, common proof metrics are not easily correlated to substantial values. We explore in Chapter 12 other proof metrics to have a better interpretation and overview on the results, which also helps to find a way to ease proofs. In the meantime, the next chapter explores the leveraged proof techniques that are refinements and evolutions from the techniques used during Pip's (MMU) formal verification.

Chapter 11

Proof techniques and evolution of Pip's formal verification process

Chapter outline

Previous chapters presented the formalisation of Pip-MPU and the high-level proofs of the security properties. These proofs relied on consistency properties and properties built up and propagated from each state modification. This chapter investigates the proof techniques that were actually leveraged to extract these properties from the micro perspective of a service or function. It also deals with the retained proof strategy from the macro perspective of all services to discover the necessary consistency properties and rapidly reach the common formal baseline of all services.

The proof development has been a personal achievement in the thesis. However, it employs formal elements and techniques inherited from Pip's proof development [100], for example Pip's proof workflow. It is intended here to clarify what has been specifically developed during the thesis and what has been directly inherited and adapted from previous works.

Fast reading track: an extensive comparison of Pip-MPU and Pip (MMU)'s proof process is presented here. The reader in a rush could be interested in the expressivity differences of the security properties 11.1.3 or the common consistency properties 11.1.4, roughly assuming the low-level proofs are the same. There is also an interest to understand the differences in the proof approach of the modification services 11.3.1. Furthermore, Pip-MPU introduces a new global proof strategy exposed in 11.4.

11.1 Formalisation

The main rationale for modifying Pip's formalisation in Pip-MPU is the lack of MMU. The intrinsic differences between MMU and MPU 2.5 imply no virtual memory and addresses, and only physical addresses. However, the notion of memory page is broader (it is a set of contiguous physical addresses) and the structural changes pass on to the services, helper functions, the expression of the properties and ultimately to the security properties to prove.

11.1.1 System state and metadata structures

At the heart of the formalisation lies the system state. It is directly borrowed from Pip (MMU). The memory state is still an association between physical addresses and memory types. The adaptation just modifies the current partition identifier to become a plain physical address instead of a memory page, which does not have much impact.

However, Pip-MPU's metadata structures are more explicit than Pip (MMU)'s.

Indeed, Pip-MPU considers five different information-rich memory types whereas Pip (MMU) considers four related information-poor types: virtual entry, physical entry,

virtual page and physical page. The entries are pointers to the pages. Whereas properties could be directly observed from the rich memory types of Pip-MPU, properties in Pip (MMU) must be inferred from the types and the context that might be very long.

Furthermore, metadata structures have similar roles and direct equivalence with C structures at implementation level, which facilitates proof understanding without deviating from the C implementation.

11.1.2 Loss of virtual memory

While memory pages could be interchanged with a single physical memory address in the system state, the notion of memory page as a memory portion causes trouble in all proofs and property expressions.

Indeed, memory pages implicitly refer to a pre-chunked memory of individual pages. A physical memory page reference is unique in the system and pages do not overlap. Plain physical addresses are not memory portions and Pip-MPU's notion of memory blocks is not really the same as memory pages. The issue is that memory blocks are locally unique in a partition but not globally: a copied block in a child partition has the same attributes as the block in the parent (except if access permissions are changed) but they both are stored in their respective kernel structures and so their block identifier is different. The virtual memory is in fact already dealing with the local identifier via virtual memory pages that are eventually resolved in unique physical memory pages. In Pip-MPU, it cannot be handled the same way with the metadata structures. However, the physical addresses are unique as they follow a strict relation order. What could be handled in the abstract notion of memory pages, can then be transformed into explicit sets of unique physical addresses.

The loss of memory pages completely shakes the proof baseline because it is deeply rooted in all memory types, even physical addresses that are tuples of type (page*index). Even where physical addresses were directly used, there must be changes.

11.1.3 Security properties

The security properties are soaked with memory pages and must then be adapted. However, the semantic should not change, because it reflects Pip's security model as a partition tree. Pip-MPU's security properties are then almost identical to Pip (MMU)'s to ensure the same properties and be confident about the semantic. The only (light) changes are that *page* is replaced by *paddr* and the name of the root partition identifier is changed from *pageRootPartition* to *multiplexer* (initially inherited from Pip that later changed the name and not modified in Pip-MPU).

```
(** PIP (MMU) **)
(** THE VERTICAL SHARING PROPERTY
 ↪ *)
Definition verticalSharing s : Prop
 ↪ :=
forall parent child : page ,
  In parent (getPartitions
   ↪ pageRootPartition s) ->
  In child (getChildren parent s) ->
  incl (getUsedPages child s)
   ↪ (getMappedPages parent s).

(** THE ISOLATION PROPERTY BETWEEN
 ↪ PARTITIONS *)
Definition partitionsIsolation  s :
 ↪ Prop :=
forall parent child1 child2 : page ,
  In parent (getPartitions
   ↪ pageRootPartition s)->
  In child1 (getChildren parent s) ->
  In child2 (getChildren parent s) ->
  child1 <> child2 ->
  disjoint (getUsedPages child1
   ↪ s)(getUsedPages child2 s).

(** THE KERNEL DATA ISOLATION
 ↪ PROPERTY *)
Definition kernelDataIsolation s :
 ↪ Prop :=
forall partition1 partition2,
  In partition1 (getPartitions
   ↪ pageRootPartition s) ->
  In partition2 (getPartitions
   ↪ pageRootPartition s) ->
  disjoint (getAccessibleMappedPages
   ↪ partition1 s) (getConfigPages
   ↪ partition2 s).
```

```
(** PIP-MPU **)
(** THE VERTICAL SHARING PROPERTY
 ↪ *)
Definition verticalSharing s : Prop
 ↪ :=
forall parent child : paddr,
  In parent (getPartitions
   ↪ multiplexer s) ->
  In child (getChildren parent s) ->
  incl (getUsedPaddr child s)
   ↪ (getMappedPaddr parent s).

(** THE ISOLATION PROPERTY BETWEEN
 ↪ PARTITIONS *)
Definition partitionsIsolation  s :
 ↪ Prop :=
forall parent child1 child2 : paddr ,
  In parent (getPartitions
   ↪ multiplexer s) ->
  In child1 (getChildren parent s) ->
  In child2 (getChildren parent s) ->
  child1 <> child2 ->
  disjoint (getUsedPaddr child1 s)
   ↪ (getUsedPaddr child2 s).

(** THE KERNEL DATA ISOLATION
 ↪ PROPERTY *)
Definition kernelDataIsolation s :
 ↪ Prop :=
forall partition1 partition2 :
 ↪ paddr,
  In partition1 (getPartitions
   ↪ multiplexer s) ->
  In partition2 (getPartitions
   ↪ multiplexer s) ->
  disjoint (getAccessibleMappedPaddr
   ↪ partition1 s) (getConfigPaddr
   ↪ partition2 s).
```

However, Pip-MPU has deeper changes that the slight cosmetic modifications above do not show. All «*Paddr*» functions hide the retrieval of all addresses contained in blocks where in Pip (MMU) it stops at the page granularity. In Pip-MPU, it is like all physical addresses where extracted from the memory pages, here the memory blocks. For example, getMappedPaddr, first retrieves all mapped blocks (all blocks of the mentioned partition), and for each of these blocks collects all their contained physical addresses in a recursive manner.

```
(** The [getMappedPaddr] function Returns all physical addresses
 ↪  contained in all present blocks of a given partition *)
Definition getMappedPaddr (partition : paddr) s : list paddr :=
let blockList := getMappedBlocks partition s in
getAllPaddrAux blockList s.

(** The [getAllPaddrAux] function Returns all addresses contained in the
 ↪  given list of memory blocks *)
Fixpoint getAllPaddrAux (blocklist : list paddr) (s : state) :=
match blocklist with
| [] => []
| block::list1 => match lookup block (memory s) beqAddr with
                        | Some (BE bentry) => getAllPaddrBlock
                         ↪  bentry.(blockrange).(startAddr)
                         ↪  bentry.(blockrange).(endAddr)
                        ++ getAllPaddrAux list1 s
                        | _ => getAllPaddrAux list1 s
                        end
end.

(** The [getAllPaddrBlock] function Returns all addresses between the
 ↪  given start and end addresses *)
Definition getAllPaddrBlock (startaddr endaddr : paddr) : list paddr :=
getAllPaddrBlockAux 0 startaddr (endaddr-startaddr).
```

11.1.4 Consistency properties

Pip-MPU's consistency properties presented in Chapter 9 were categorised in 3 categories: 1) inner partition purely structural, 2) inner partition set structural, 3) inter-partition set. The same categorisation can be done in Pip (MMU). Because of the enriched metadata structures, the two first categories are similar but quite different from Pip

(MMU). Pip (MMU) must colour each memory entry with a meaning by a thicker context, for example with `wellFormedFstShadow`.

```
(** PIP (MMU) **)
Definition wellFormedFstShadow (s :
↪  state) :=
forall partition,
In partition (getPartitions
↪  pageRootPartition s) ->
forall va pg pd sh1,
StateLib.getPd partition (memory s)
↪  = Some pd ->
StateLib.getFstShadow partition
↪  (memory s) = Some sh1 ->
getMappedPage pd s va= SomePage pg ->
exists vainparent,
↪  getVirtualAddressSh1 sh1 s va =
↪  Some vainparent.
```

```
(** PIP-MPU **)
Definition
↪  wellFormedFstShadowIfBlockEntry
↪  s :=
forall pa,
isBE pa s ->
isSHE (CPaddr (pa + sh1offset)) s.
```

The third category is even more similar, almost identical to Pip (MMU), because it deals with the relationships between partitions which should be captured in Pip-MPU as well, for example with `isChild`.

```
(** PIP (MMU) **)
Definition isChild  s :=
forall partition parent : page,
In partition (getPartitions
↪  pageRootPartition s) ->
StateLib.getParent partition
↪  (memory s) = Some parent ->
In partition (getChildren parent s).
```

```
(** PIP-MPU **)
Definition isChild  s :=
forall partition parent : paddr,
In partition (getPartitions
↪  multiplexer s) ->
pdentryParent partition parent s ->
In partition (getChildren parent s).
```

11.1.5 Proof framework

The proof framework is completely reused from Pip (MMU) with local adaptation for Pip-MPU.

The proof goal is obviously the same: the isolation invariant on each service by Hoare logic, even with the mentioned differences in the security and consistency properties. This means the Hoare triples to prove and their Coq formal expression are the same.

Moreover, the instruction-by-instruction unfolding, which leads to a chain of Hoare triples to prove, is identical with the same use of weakest preconditions and invariants.

Also, the general service pattern is identical: a check phase supplemented by a modification phase for modification services. The check phase reveals the memory state at time of execution and, if successful, engages the modification stage. Indeed, the user can invoke the services at any time. Only the kernel components are interesting for Pip and not the content of user memory blocks, because Pip only manipulates the metadata structures to ensure the partition tree. However, Pip has no notion of the context when a service is called. The check phase rejects any parameter mismatching the expected context to run the modifications (for example references to blocks that do not exist). The context must be fully revealed in order to connect with the pre-conditions of the modification instructions and is used to validate the state consistency (consistency properties). The check phase also facilitates the proof by exiting the function without any state modifications and thus the risk of an inconsistent state for the security properties.

Furthermore, Coq requires a termination proof for all recursively defined functions. In Pip (MMU), this is done by announcing a *fuel* parameter, a natural number, that decreases at each recursive call and thus terminates at some point. This satisfies Coq, however, any state invariant useful for Pip's proof might not be true if the recursive call ends too quickly due to a too small fuel. And that parameter must have some meaning to connect to the implementation, *i.e.* the reality of the algorithms. Pip (MMU) usually sets the fuel to the maximum number of memory pages in the system: no search can exceed this number. Similarly, Pip-MPU sets the fuel to the maximum number of physical addresses in the system. For example, for a search in a linked list without cycles, and the information that all elements are unique, the fuel is enough to cover any number of elements in that list, because pointers between each element are no more than the number of physical addresses in the system's memory.

In addition to that, all helper functions had to be adapted due to the switch from memory pages to physical addresses and the structural differences (no virtual memory so no translation tables in the structures, no optimisation structures, poorer memory types). However, they do have similar structures and are used to retain properties in the same way.

Finally, the principle of having consistency properties that serve the proofs of security properties is also transmitted from Pip (MMU).

11.2 Low-level local proofs (HAL)

All low-level proofs can be slightly adapted to Pip-MPU. Indeed, the system state is very similar that makes low-level operations (the HAL) similar as well. For exam-

ple, in readAccessible and equivalent Pip-MPU readBlockAccessibleFromBlockEntryAddr, the proof script is almost identical, with local variations because of differences in the metadata structures and property expressions (*entryUserFlag* and *bentryAFlag*).

```
(** PIP (MMU) **)
Lemma readAccessible (table : page)
 ↪  (idx : index) (P : state -> Prop)
 ↪  :
{{ fun s => P s /\ isPE table idx s
 ↪  }} MAL.readAccessible table idx
{{ fun (isaccessible : bool) (s :
 ↪  state) => P s /\ entryUserFlag
 ↪  table idx isaccessible s }}.
Proof.
eapply WP.weaken.
apply WeakestPrecondi-
 ↪  tions.readAccessible .
simpl.
intros.
destruct H as (H & Hentry).
apply isPELookupEq in Hentry
 ↪  ;trivial.
destruct Hentry as (entry & Hentry).
exists entry. repeat split;trivial.
apply lookupEntryUserFlag;trivial.
Qed.
```

```
(** PIP-MPU **)
Lemma readBlockAccessibleFromBlock-
 ↪  EntryAddr (paddr : paddr) (P :
 ↪  state -> Prop) :
{{ fun s => P s /\ isBE paddr s  }}
 ↪  MAL.readBlockAccessible
 ↪  FromBlockEntryAddr paddr
{{ fun (isA : bool) (s : state) => P
 ↪  s /\ bentryAFlag paddr isA s }}.
Proof.
eapply WP.weaken.
apply WP.getBlockRecordField.
simpl.
intros.
destruct H as (H & Hentry).
apply isBELookupEq in Hentry
 ↪  ;trivial.
destruct Hentry as (entry & Hentry).
exists entry. repeat split;trivial.
apply lookupBEntryAccessible-
 ↪  Flag;trivial.
Qed.
```

11.3 High-level local proofs (services)

In this section, we take a look at the proof conduct from the service perspective. I first compare the proof approaches in Pip (MMU) and Pip-MPU. While the approach is the same for pure informational services, the code and proofs co-design approach has not been followed in Pip-MPU, which leverages instead a technique based on list properties propagation, proof checkpoints and full low-level proof modularity. I then specifically describe the iterative procedure engaged to prove the services in Pip-MPU.

11.3.1 Proof approach differences

First of all, from a general perspective, the proof script patterns are the same, as illustrated in Listing 13.

Listing 13 Typical proof pattern used in Pip (MMU) and Pip-MPU.

```
(* isolate the next instruction *)
eapply WP.bindRev.
{  (** weaken the hoare triple *)
   eapply weaken.
   (** apply the invariant corresponding to function FFF *)
   apply Invariants.FFF.
   (** Prove the current properties link to the applied invariant *)
   intros. simpl.
   (* ... *)
}
```

Also, the pure informational service proofs are directly inherited from Pip (MMU). The goal is to propagate the initial security and consistency properties all along the instruction-by-instruction proof steps with no state modification.

On the contrary, the proof style differs for modification services.

One particular difference is the handling of temporary state, in the middle of the service, where consistency properties might not be all true. For example, this happens when an instruction modifies a field which is linked with another one in some of the consistency properties. The temporary state makes the linked consistency properties inconsistent. When reaching the end of the service, the temporary state resolves to be consistent (or should do, the proofs ensure this). Where Pip (MMU) invoked co-design to flip instruction order to respect the consistency at all times, Pip-MPU introduces the concept of **proof checkpoints**. These checkpoints are instants in the proofs where all consistency properties are true. Consistency properties are then only proved when required and not at each instruction. Obviously, the high-level service's pre-and post-conditions include all consistency properties to start and end in consistent states.

This is illustrated with function invariants. Indeed, Pip-MPU pulls the modularity to a more extreme level than Pip (MMU) and tries to gather in a same function any similar operation (same instructions). The principle is that reusable functions in the code can also be reused with proven function invariants as proof bricks in the proof script. For example, the function `insertNewEntry` inserts a new entry in the *Blocks* structure by filling the fields with the given parameter. This function is used in the `addMemoryBlock` and `cutMemoryBlock`, to insert a new entry respectively in a child partition or in the current partition. In the *cut* case, consistency properties are all true after the insertion. Yet, this is a partial operation for `addMemoryBlock` case, because the inserted entry must be referred in the parent to be consistent (consistency property *sharedBlockPointsToChild*) which is not done inside this function to generalise over

the *cut* operation. Thus, only a sub-set of the consistency properties can be propagated (*consistency*1). In Pip (MMU), this would have been solved by another instruction order to find a consistent configuration. However, this new method allows a fast pace proving without going back to the design phase, especially if the instruction order made sense for the system developer at first. In Pip-MPU, consistency properties are split in two sets: *consistency*1 and *consistency*2. The first set *consistency*1 is composed of consistency properties that have been true in all encountered function invariants, whereas the second set *consistency*2 relates to different structural elements and so only true after some consistent state modifications, especially at the proof checkpoints.

Some properties are nevertheless always propagated at each instruction, in particular list-related properties. Lists are required for all consistency property types and are complex to build out of a state that has been modified several times. They are then constructed instruction-by-instruction by lemmas specifically crafted for the modification. For example, the lemma `getChildrenEqPDT` proves the *partition*'s children have not changed when modifying the memory state with the insertion of a *PD* structure at address *addr'*. To invoke the lemma and prove the equality, the conditions are: there was a *PD* structure entry at the same address *addr'* in the previous state, the structure field has not changed in the new *PD* structure, and the *StructurePointerIsKS* consistency property was true at the previous state.

```
Lemma getChildrenEqPDT partition addr' newEntry s0 pdentry0:
lookup addr' (memory s0) beqAddr = Some (PDT pdentry0) ->
(structure newEntry) = (structure pdentry0) ->
StructurePointerIsKS s0 ->
getChildren partition {|
                  currentPartition := currentPartition s0;
                  memory := add addr' (PDT newEntry)  (memory s0) beqAddr
        |} =
getChildren partition s0.
```

Note the requirement to prove *StructurePointerIsKS* at the previous state in order to invoke that lemma. All other consistency properties are not required. However, the combination of all lists (children, partitions, memory blocks, free slots) sets the minimum baseline for the consistency properties to prove at each instruction. The order of the instructions and the importance of the modification (*e.g.* changes to the PD flag in the *Shadow*1 structure imply many lists and structural changes) determine the minimum set of necessary consistent properties to propagate.

11.3.2 Procedure to prove a modification service

We have described in the previous sections the different low-level and high-level lemmas invoked with a special pattern and leveraging the checkpoint technique. We interest us now in the timing and the followed iterative procedure to prove a new service. We take a generic example of a modification service and present some tricks to ease the proofs and accelerate the verification. The main spirit is to rapidly span the whole proof without proving all elements, and then come back to prove the missing elements by increasing the complexity of the proofs at each iteration. It is a fail-fast approach with fast identification of bugs conflicting with the security properties.

Preliminary analysis The service should be clearly understood and the expected memory state (notably lists) at service completion must be described. The metadata structures and modified fields engaged in the service should be listed and should match the service understanding. Based on the expected memory state and intuition, an informal proof should be done to prove the security properties. The latter phase highlights potential misses in the expected memory state and eliminates evident bugs.

The check phase The check phase usually happens once because the underlying lemmas are simple to prove on-the-go. It consists in applying the proof pattern 13 and propagate all properties at each step.

The HAL primitives and the functions composed of checks are proved whenever encountered via dedicated lemmas. However, the post-conditions of functions might not be completely defined at that moment. Later, when an instruction requires a property that should have stemmed from that function, the corresponding lemma must be enhanced with the missing property.

The modification phase The modification phase is first simply approximated by writing down the instructions on a paper or white board, and explicitly writing the expected property for each instruction that is used in the next steps.

Next the proof pattern is applied again with the plain HAL proofs when required and mock lemmas for functions. All properties are also propagated, associated with the current state, and proof obligations are left aside for the moment (*admit* tactic in Coq). Mock lemmas are used to reach the end of the proof so to reach *the consistency and security properties proofs*. The important properties to propagate are the final modified state and all temporary states in between that will be used for the final proofs. Hence, they must be crossed once to extract the state properties.

Thereafter, we prove the security properties, isolated from the rest. They are proved

with assumed consistency properties and based on the propagated properties (including the modified state) and the projected modifications of the lists. The informal intuition of the preliminary analysis is set up and proofs should end successfully.

In the case of missing properties, we must find which instruction reveals them. Maybe the properties are there but must be combined with other ones, or they have not been propagated correctly, or they had not been deemed necessary to be included and should be proven. They must then be propagated all the way along to finish the proofs.

Once the security properties are proven, we iterate again from the first modification instruction. The goal here is to prove all simple assumptions that were used previously. Simple consistency properties, without lists, must be proven when required by the proof obligations.

Internal functions pre- and post-conditions should be adapted to their context. As said earlier, some consistency properties might not pass the function because of an intermediate state, or might not link to the function pre-condition. This is the time to identify the proof checkpoints and delay the full proofs of the consistency properties to these points.

In a last iteration, the lists are propagated and their properties proved when appropriate. This enables to prove the more complex consistency properties involving lists. Lists must eventually be identical to the projected lists used to prove the security properties. At the final step, all properties should connect with the context required by the security property proofs.

11.4 Global proof strategy: horizontal and vertical exploration strategies

We take now a holistic view on the proof process.

The overall goal is to prove all services with the techniques exposed above. They assumed a knowledge of all consistency properties and lists used to prove the security properties. This raises the question of the discovery of the consistency properties. Creating consistency properties for all logical connections between kernel elements would not be efficient because solely the properties having a purpose in the security properties proofs matter really. Pip-MPU's proof workflow introduces the **horizontal and vertical exploration strategies**.

11.4.1 Horizontal exploration

The intuition of this strategy is that we should cover most of the proofs to unveil the consistency properties really important for the proofs of the security properties.

We demonstrated, already confirmed by previous experiences with Pip (MMU), that the check phase is easier to pass than the modification phase. Hence, the horizontal exploration strategy passes all check phases in **all** services. This brings up most of the structural consistency properties.

11.4.2 Vertical exploration

The next intuition is that the more services we finish, the more properties are discovered and will be used in the proofs of the other services. We then choose a service and finish the proofs. We select a modification service because pure informational services do not bring up more properties than the first exploration.

Propagation of the lists and modifications due to state changes rely on the structural consistency properties brought by the horizontal exploration.

In a final step, the security properties proofs conclude the discovery of missing properties to terminate the proofs.

11.5 Discussion

11.5.1 Direct inheritance from Pip (MMU)

The proof workflow and much of the formalisation have been directly inherited from Pip (MMU). It shows the successful application of the proof workflow, at implementation level without refinement, for another project.

There was no need for a deep change in the formalisation because the proof goal stays the same. The memory state does not need to represent the registers or to know in which privileged mode the code executes (like in ProvenCore [29]), because the system traps in privileged mode to execute the services and the address level is enough to express all properties.

However, we have seen previously that even if the system state looks similar, which makes low-level proofs reusable, the lack of memory pages (and associated translation tables) causes huge and deep changes to all helper functions that manipulate the meta-data structures, such as entry read, storage and search. Furthermore, some services have been simplified, for example `createPartition` that only sets up the PD structure instead of also setting up the MMU, Shadow 1, Shadow 2 and LL structures. In addition to that, even if Pip-MPU has some common grounds with Pip (MMU), for example the setting of the PD flag in `addMemoryBlock`/`addVAddr`, the extreme modularity pursued by Pip-MPU and the lack of virtual memory did not match the same instruction order or internal functions. This led to (almost) full refactoring of the services and make Pip (MMU)'s high-level proofs useless for Pip-MPU. This is far from the expectations at the

beginning of Pip-MPU's design and the maximisation of proof reuse. Moreover, Pip (MMU)'s proofs were too deeply rooted in the concept of memory pages, so that only the low-level proofs could be reused with light changes.

11.5.2 Pip-MPU's local and global proof strategy

This work's goal was two-fold: rapidly identify the consistency properties and to prove a modification service.

The local approach with checkpoints and the global proof strategy of horizontal and vertical explorations made this possible, however, deviated from the sequential approach taken by Pip (MMU). The consequence is that services are no more preemptible because we allow inconsistent states which ultimately resolve in a consistent one. Instead, Pip (MMU) always ensured consistent states by flipping the instruction order, part of the co-design process. Pip-MPU's design has been frozen and thus the adopted approach avoids coming back to the design stage. Furthermore, Pip's services, in Pip (MMU) and in Pip-MPU, are actually not preemptible. This is ensured by disabling the interrupts before the service treatment and by enabling them again after. The approach has then no real impact.

Other methodologies could have been used to discover the most important consistency properties for the security properties proofs.

One of them is a preliminary analysis of the security properties themselves. They dictate what information is required to verify them. The properties announce the lists of children, partitions, mapped memory blocks and accessible mapped memory blocks. A finer grain analysis also identifies the kernel elements that are crucial for the security properties proofs, like the mapped blocks list, the PD flag or PDchild fields, the addresses contained in the blocks, the metadata structures, and the accessible flag. Consistency properties intervene to get the status of all these elements at the end of the service execution.

11.5.3 Vertical exploration

This work covers the vertical exploration of `FindBlock`, `readMPU` and `addMemoryBlock` in that order.

The two first services are the only pure informational services of Pip-MPU's API. Their proofs are therefore relatively easy to conduct that explains why we choose to cover them first.

On the contrary, `addMemoryBlock` is a modification service that affects the whole proof process. It had many characteristics that were interesting to test: the function invariant, it modified all the memory types which touch almost all structural consistency

properties, it modified the lists (free slots, mapped blocks and accessible mapped blocks) by adding and removing elements, it gave memory to a child which challenges the Vertical Sharing and the Horizontal Isolation security properties. Hence, it outlines the proof process for all modification services.

For the next modification service to prove, it would be interesting that it involves this time kernel structures and recursive internal functions like `deletePartition` that would also demonstrate modifications of the partitions and the children lists. Some proofs would be facilitated by that, for example, all consistency properties involving duplicates, whereas others, typically the recursive functions are expected to be more difficult.

11.5.4 Harmonise Pip (MMU) and Pip-MPU's proofs

Pip-MPU's proofs largely deviated from Pip (MMU), even if Pip-MPU inherits elements of the proof baseline. The main failure to harmonise the two projects are the memory pages/memory blocks. Yet, both notions seems close, as later discussed in Section III.

From the proof perspective, Pip-MPU already had to lower down the concept to the physical addresses contained in the memory blocks, instead of dealing directly with the memory blocks like Pip (MMU) uses the memory pages. This was necessary to keep the expressions of the security properties similar. It seems possible to lower down the concept of memory pages in the same way, to the grain of the physical address.

Another approach could be to generalise the concepts of memory blocks to embrace memory pages in Pip (MMU). A memory page would be a memory block with constraints on the size and lists of memory blocks should be generalised to not depend either on the MMU or the MPU constraints. Pip-MPU security properties would then be more similar to Pip (MMU)'s. And Pip (MMU) could also be rewritten with enriched memory types that would join Pip-MPU's proof pattern and consistency properties.

While these trials would harmonise the proof baseline, and bring the security properties proofs close, the services' algorithms and memory types differences would still end in completely different proofs, if Pip (MMU) and Pip-MPU do not conciliate the two approaches on a more abstract level, which would maybe change the proof approach to a refinement strategy.

11.6 Conclusion

Pip-MPU mobilises new proof approaches while relying on the proof workflow and formalisation of Pip (MMU). In particular, it leverages novel local and global proof strategies (proof checkpoints and horizontal/vertical explorations).

The aim of Pip-MPU's design has always been to facilitate the proof of the security properties (*cf.* Section III for more insights). Structural convergence and expressions of the security properties at the grain of physical addresses also participated in proof similarities with Pip (MMU). The low-level proofs on the HAL primitives could be adapted with light modifications. Unfortunately, because of code and memory types differences, none of the high-level proofs of Pip (MMU) could be reused for Pip-MPU.

Pip-MPU also pursued to accelerate the proof process with novel proof techniques. It enables to freeze the design phase when verification starts, which means it does not follow the co-design approach of Pip (MMU). Consistency properties are only proved when required and split into two different subsets that gather at the proof checkpoints. Combined with horizontal and vertical exploration strategies, it permits to rapidly find the important properties and shape the proof baseline. Later vertical explorations might move the consistency properties sets and could even be declined in more subsets if necessary.

As with Pip (MMU), the proof process is very cumbersome with no automation. Also, many duplicate proof script portions are reported and could certainly be generalised with more efforts; however, like with Pip (MMU), the main goal is to get the proofs and not their prettiness. Nevertheless, Pip-MPU tried to clarify the proof process for later maintenance and verification with a clearer proof script organisation and a higher level of modularity.

Chapter 12

Proof monitoring and proof path

Chapter outline

With past formally verified projects, there are two certainties: proof development is more demanding than code development, and proof engineering is a recent research field. Indeed, last chapter gave us some measures of the demanding proof effort of Pip-MPU. Furthermore, one encountered issue with Pip-MPU was the inability to communicate proof development progress to my line of hierarchy only used to software development processes. It implied the presentation of the proof status and an analysis of the proof direction which should, ideally, show an optimised proof path.

State-of-the-art metrics were not enough for these two goals. This chapter goes beyond what is currently proposed in the literature by the introduction of new proof metrics to quantify proof development and proof effort, to compare different proof designs, and to select the best effort proof path. We also propose a novel approach to monitor the proof development process.

Fast reading track: Many original metrics are presented in this chapter, and they are mostly chained together. Anyways, I would like to stress the effective reusability metric 12.2.4 and the computation of the proof complexity 12.3.4 as a combination of the impact score of an instruction on the properties (its disturbance level) with the inherent proof complexity of these properties. All these metrics contribute in optimising a global proof path strategy 12.4.1 that selects the best proof direction. We illustrate this with Pip-MPU. Furthermore, the proof development is closely monitored by a dynamic proof dashboard 12.5.

12.1 Existing proof metrics

The need to quantify the proof activity comes from the desire to assess the proof development, to compare it with other verification projects, and to optimise the development to accelerate it while reducing the proof effort.

The proof goal is to develop the proof elements that form the proof bricks for a project's formal verification.

12.1.1 Proof element

Definition 12.1.1 (Proof element)**.** A proof element is an atomic statement, *i.e.* theorems, lemmas and propositions, with its respective proof.

Indeed, proof elements refer to each other to build upon previous proved statements with the aim of proving a global statement, like Pip's security properties for all services.

12.1.2 Metrics on proofs and metrics on the proof process

I distinguish two main categories of metrics: about the proofs themselves and about the proof process. The latter is closely linked to the former.

Common metrics for proofs in formally verified projects [85, 109, 102] are the number of Lines of Proof (LoP), number of theorems, number of invariants, number of discovered bugs, and amount of verified code. We have already presented these metrics concerning the proofs in the last chapter, as well as additional metrics such as the amount of RAM required to conduct the proofs and the number of Coq tactics.

Some works studied relationships between these metrics, and notably found a linear correlation between the proof effort and the proof size or a quadratic correlation between a property's size and the corresponding proof script in the seL4 projects, which depend on the seL4 proof-style [167, 122].

To speak about the proof process, we first define the proof effort:

Definition 12.1.2 (Proof effort)**.** The proof effort gives a measure of the amount of work necessary to complete a proof (human effort).

In formally verified projects, the proof effort is usually given in person-time (days, months, years). The proof process is impacted by the proof approach, for example refinement at various granularity and with different number of abstraction layers, which leads to different meaningful metrics to monitor the considered projects [5]. Other works [21] try to identify proofs characteristics in order to quantify the proof effort (number of theorems weighted by complexity, theorem size and number of dependencies) or to detect potential issues (depth of the dependency tree and number of child theorems that could complicate maintenance, similarity score between theorems revealing a bad design).

Hence, the set of meaningful formal metrics is not yet fixed and depends on the project. Current metrics fall short in giving a precise overview of the proof development status and do not consider different proof strategies. In the rest of this chapter, we present new metrics to better understand the proof process in Pip-MPU and the relationships between proof elements in the objective to report the proof status and select the best proof strategy.

12.2 Code and proof relationship

Like **code elements** (function, module, instruction) are assembled in code, so are proof elements during the proof process. In Pip-MPU, as demonstrated in the previous chapters, the approach is to freeze the design and to lean on the correspondence between

each function or primitive and their respective proof element. Each function or primitive has a main proof element and inner proof elements that help the proof development such as property extraction or property equivalence. As a consequence, the number of main elements, *i.e.* the proof bricks, is known in advance once the design is frozen.

Furthermore, it results the number of code elements (functions and primitives) indicates the minimum number of proof elements.

Theorem 12.2.1 (Minimum number of proof elements). For the set of code elements composed by the subset of functions $F = F_1, F_2, ..., F_f$ and the subset of primitives $P = P_1, P_2, ..., P_p$, and for the set of corresponding proof elements $PE = PE_1, PE_2, ..., PE_l$, then

$$|F| + |P| <= |PE|$$

With corresponding code and main proof elements, the code and proof dependency graphs match from a high-level perspective. The dependency graph is a polytree, a type of directed acyclic graph which underlying undirected graph is a tree. It essentially traces the services' subroutines, and stops above the low-level primitives, *i.e.* functions have at least two primitives. It is a subset of the call graph and control-flow graph without representing the recursive calls or the route back from a subroutine to the main function.

Definition 12.2.1 (Dependency graph)**.** The dependency graph of code, respectively proofs, is defined as the set $G_{function} = (F, DC)$, respectively $G_{proof} = (PE, DP)$, where the set of functions $F = F_1, ..., F_n$ are the nodes in $G_{function}$, respectively where the set of main proof elements $PE = PE_1, ..., PE_l$ are nodes in G_{proof}, and the set of dependency DC are arrows (edges) in $G_{function}$, respectively DP in G_{proof}.

12.2.1 Code and proof reuse

Relying on code reuse is the best way to also create reusable proof bricks. Of course, proof bricks must be general enough to embrace most of the cases where the corresponding code element is used. Proof reuse avoids similar proofs at different instants in the proof script. The strategy to develop reusable lemmas or heuristics like some tactics (proof hints) to reduce the proof effort is still researched nowadays [94]. Thus, a modularity of proof elements mapped on the modularity of code elements represents the minimal proof effort. In the case that proofs do not follow the code modularity, proof effort is increased by as many components that are not reused but could be. On the contrary, the more reused elements there are, the less development is needed (we consider the benefits of reusability superior to the resources consumed to generalise the elements to become reusable). We are then interested to compute the actual ratio of reused elements on all nodes of the dependency graph.

To compute this ratio, we define the main reused element subgraph and the full reused element subgraph of the dependency graph $G = (E, D)$, where $E = E_1, ..., E_n$ is the set of (code or proof) elements and D the set of dependency arrows (edges).

Definition 12.2.2 (Main reused element subgraph)**.** Main reused elements are nodes in the dependency graph G which have multiple incoming edges (the indegrees). The subgraph of reused elements is $R = \{G' \in G | \forall E_i \in G[E], deg^-(E_i) > 1\}$.

Definition 12.2.3 (Full reused element subgraph)**.** Any subroutines of reused elements are obviously also reused, but might not appear in R if they have a unique incoming edge. The full reused element subgraph is then R augmented by all elements in the subtrees $S = S_1, ..., S_s$, $s \in \mathbb{N}$, rooted at a main reusable element, so $G_{element\_reused} = \{G' \in G | \forall G_i \in G, \forall R_j \in R, \forall S_k \in S_{R_j}, G_i[E] \in S_k \wedge deg^-(G_i[E]) = 1\} = (E_{element\_reused}, A_{element\_reused})$.

Definition 12.2.4 (Reused elements ratio metric)**.** The reused elements ratio RR metric is the ratio of the number of reused elements considering all elements in the dependency graph. It is the ratio of the number of reused elements in $E_{element\_reused}$ over the total number of elements in the dependency graph G:

$$RR = \frac{|E_{element\_reused}|}{|E|}$$

Note that service root nodes are never in $E_{element\_reused}$ because they do not have any incoming edges.

Reused ratio in Pip-MPU

In Pip-MPU, the modularity of code is translated in proofs, so $G_{function} = G_{proof} = G$. For the sake of clarity and readability, we only model in Figure 12.1 elements containing write instructions (the full dependency graph should also contain any read instructions related elements). In this figure, reused elements are highlighted with the blue boxes. Thus, the more blue boxes, the more reuse.

For Pip-MPU, the reused ratio, given the simplified dependency graph depicted in Figure 12.1, is:

$$RR = \frac{10}{36} \approx 27.7\%$$

12.2.2 Reused proof difference

In some situations, code and proof dependency graphs might not match, for example because function-level lemmas are difficult to set-up or because the functions are small and the proof effort is not worth it. It results in a difference between a potential full

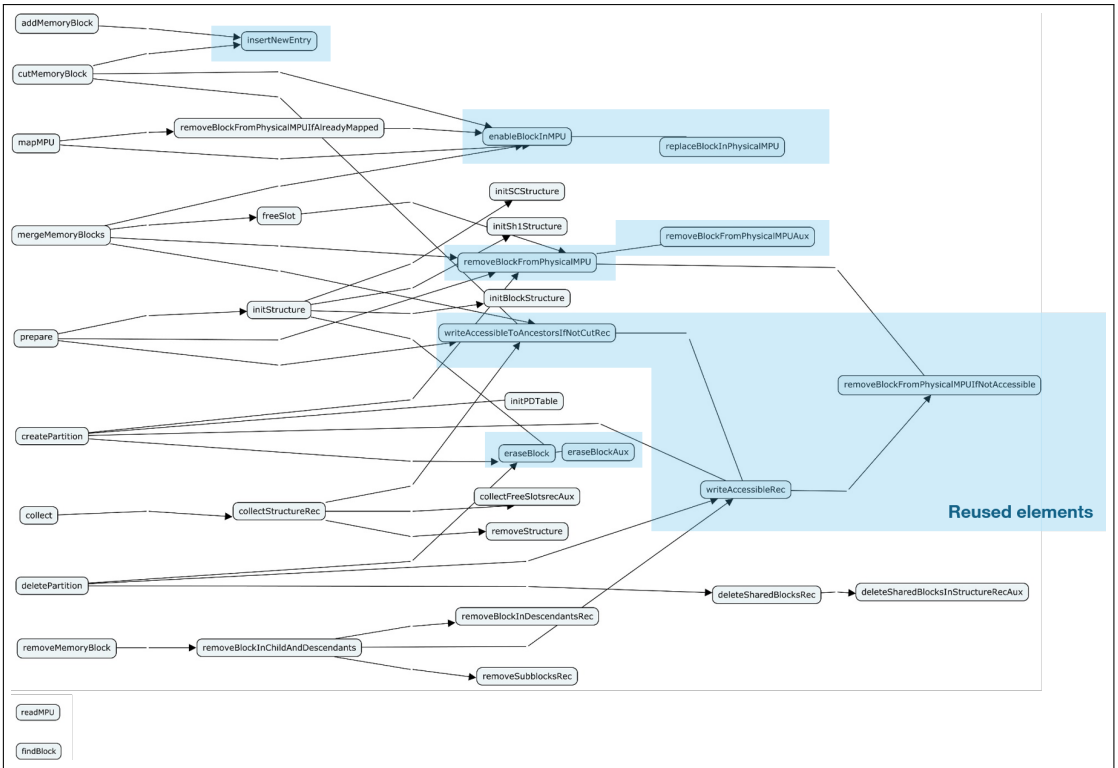


Figure 12.1: Pip-MPU dependency graph. Service entry points are on the left. Reused elements are highlighted in the blue boxes.

reusability of elements if dependency graphs match (equal to a full code reuse RR_{code}) and the actual proof reuse RR_{proof}.

Definition 12.2.5 (Reused proof difference)**.**

$$RR_{code} - RR_{proof}$$

The higher it is, the more it indicates missed opportunities for optimisation, ease of proof design, duplicate proofs by creating useless additional lemmas or by writing similar patterns in the proof script.

12.2.3 Reused proof difference in Pip-MPU

In Pip-MPU, there are no differences between the actual proof reuse RR_{proof} and the code reuse RR_{code}, so the reused proof difference is 0. Thus, the proof process leverages the full potential of modularity and shows no additional useless proof efforts. Indeed, in the case of Pip-MPU, the proof developer is more interested in reducing the proof effort than having the most compact and elegant proofs at the expense of demanding efforts.

12.2.4 Effective reusability

The counterpart metric to reused ratio is its effective reusability. In other words, each time an element is reused, it increases the metric. It is a measure of the entanglement of all the services. The more elements are present in other services, the higher the effective reusability metric.

To compute this metric, we need to define the unfolded dependency graph, the no-reused-element-pruned subgraph and the reused-pruned subgraph.

Definition 12.2.6 (Unfolded dependency graph)**.** The unfolded dependency graph is defined as $G_{unfolded} = (F', A')$ which is G without any modularity with F' and A' being respectively the new set of functions and dependencies. It is like following each dependency path from the root nodes by copy-pasting all reused elements for each path, the worst case for code/proof development and maintenance. The total number of elements to develop is $|F'|$. $|F'|$ is equal to the total number of elements in each subtree $SG' = SG'_1, ..., SG'_s \subseteq G$ rooted in the service nodes, so $|F'| = \sum_{i=1}^{s} |SG'_i|$.

Definition 12.2.7 (No-reused-element-pruned subgraph)**.** The no-reused-element-pruned subgraph is the code dependency subgraph G_{pruned} which is G pruned away from reused elements. That is, we are interested in $G_{pruned} = G \setminus G_{element\_reused}$. We relate to its subtrees as $SG'' = SG''_1, ..., SG''_s \subseteq G_{pruned} \subseteq G$ rooted in the service nodes.

Finally, we can define effective reusability. The effective reusability gives the frequency of appearance of the elements in the polytree G, *i.e.* the global reusability of elements given any subtree of G rooted in the services.

Definition 12.2.8 (Effective reusability metric)**.** We consider the polytrees $G_{element\_reused}$, G_{pruned} and any of its subtrees $SG'' = SG''_1, ..., SG''_r \subseteq G_{pruned} \subseteq G$ rooted in the service nodes ($r \in \mathbb{N}$), and $G_{unfolded}$ with any of its subtrees $SG' = SG'_1, ..., SG'_s \subseteq G$ rooted in the service nodes ($s \in \mathbb{N}$). Because the rooted subtrees depend on the same service nodes, we have $r = s$.

Then, the effective reusability ER metric is defined as:

$$ER = \frac{|G_{element\_reused}| + \sum_{i=1}^{r} |SG''_i|}{\sum_{i=1}^{r} |SG'_i|} = \frac{|G_{element\_reused}| + \sum_{i=1}^{r} |SG''_i|}{|F'|}$$

ER gives the rate of proof elements to prove with reuse compared to no reuse, and so $1 - ER$ gives the spared ratio of proof elements thanks to reuse.

Indeed, reused elements have a one-time design or verification cost such as they just need to be counted once, hence $|G_{element\_reused}|$ which is added to the total element usage.

Note that the order of considering each dependency path of G_{pruned} from the root nodes (each subtree) has no influence on the result.

In the following, we relate to the ER's dividend as U and ER's divisor as E; so $ER = \frac{U}{E}$.

12.2.5 Effective reusability in Pip-MPU

We illustrate the benefit the effective reusability metric with the Pip-MPU memory management services.

For Pip-MPU, we study each service subtree and compute the values reported in Table 12.1. We obtain an effective reusability of 50% which indicates a reduction of half of the number of proof elements to actually prove. In other words, Pip-MPU largely relies on reusable elements, which directly benefits the proof effort by reusing formerly proved lemmas.

12.3 Proof complexity: fine-grained analysis of properties and code elements

The previous section demonstrated the benefit of reusability on proofs. We now study the complexity of each proof element that helps to compute an overall estimation of the proof effort to provide and to decide an overall proof strategy.

Service ($r = 11$)	$\lvert SG''_i \rvert$	$\lvert SG' \rvert$
addMemoryBlock	1	2
cutMemoryBlock	1	7
mapMPU	2	6
mergeMemoryBlocks	2	13
prepare	5	13
createPartition	2	9
collect	4	7
deletePartition	3	7
removeMemoryBlock	4	6
readMPU	1	1
findBlock	1	1
Total sum	26	72
U = $\lvert G_{element\_reused} \rvert$ + 26 = 10+26 = 36		
E = |F'| = 72		
Effective reusability : $\frac{U}{E} = \frac{36}{72} = 50\%$		

Table 12.1: Effective reusability metric table for Pip-MPU.

We take the example of Pip-MPU to illustrate our proposal. The objective is to be able to evaluate the complexity of the isolation invariant proofs by analysing the instructions composing the service. The intuition here is that the complexity of a proof depends on the impact of the instructions on the properties of the isolation invariant (some instructions disturb more the invariants than others) and how difficult these properties are to prove.

12.3.1 Proof impact score at the primitive level

The first proof brick in the proof process is the primitive. We have seen previously that read instructions do not cause any particular difficulty because the memory state does not change. However, write instructions do change the memory state.

In the case of write instructions, the impact on the proofs depends on the modification. For example, in Pip-MPU, a modification of the access permission rights are not decisive for the isolation invariant: it does not impact the security properties. In the contrary, if the PDflag field of a Shadow 1 entry is modified, it inherently means the corresponding block holds a PD structure (a child identifier) according to the consistency property `PDTIfPDFlag`. This means there is a new child and so the partition tree is modified which might invalidate the security properties.

We easily identify the most determinant fields for the security properties, notably the PDflag, PDchild, the memory types (PDT, BE, SHE, SCE), and the accessible flag.

More than that, we expand the analysis to all the fields of the kernel metadata

structures. We classify which modifications impact the isolation invariant properties the most by an analysis of the fields used in the properties. For that, we first identify which memory types or fields are used in the properties and bind them together, as illustrated in Tables [12.2, 12.3, 12.4, 12.5].

Fields and memory types are extensively used in lists present in the security properties and some of the consistency properties. It is possible to reflect the impact of a modification on a specific list at the modification instruction.

If we take the previous example again by setting the PDflag field of some Shadow 1 entry in a specific partition, then the list of children of that partition extends by one and so does the partition tree list. We can then create a lemma that reports the list modification and apply it at the modification instruction to propagate the new lists. Hence, given some amount of work to create this lemma, the modification does not disturb the property as much as foreshadowed anymore. Such possibility is marked by a '*' in the tables.

There will still be some proof work to adapt to local situations and locally prove consistency properties, however on a unique state modification (contrariwise to the proofs of the consistency properties at function level that include many state modifications). The presence of '*' informs that a certain amount of proof work must be done once, and then less each time it is encountered. Thus, they are counted separately.

The same lists are used across different properties and they are limited: `getPartitions`, `getChildren`, `getMappedBlocks`, `getAccessibleMappedBlocks`, `getConfigBlocks`, `getMappedPaddr`, `getConfigPaddr`, `getUsedPaddr`, `getAccessibleMappedPaddr`.

Table 12.6 resumes what impact is expected on the proof of properties by the modification of a single instruction.

From Table 12.6, and given lemmas for list modifications, we conclude that any modifications of the present flag and PDflag, consolidated by the score on *BE* and *PDT* types, impact the properties the more, and by extension the proofs. On the contrary, as expected, modifications of the access permissions do not disturb the properties very much.

Each field modification is associated with the instruction operating the change. In Pip-MPU, one primitive changes one field at a time, so we call the **field change impact matrix** also the **instruction impact matrix** (*IIM*).

The more instructions and the more write instructions there are, the higher the impact score is, and higher is the proof complexity.

	Properties	PDT type	BE type	SHE type	SCE type	PADDR type
Consistency	nullAddrExists	-	-	-	-	1
	wellFormedFstShadowIfBlockEntry	-	1	1	-	-
	PDTIfPDFlag	1	1	1	-	-
	AccessibleNoPDFlag	-	1	1	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	1	1	1	1	1
	multiplexerIsPDT	1	-	-	-	-
	currentPartitionInPartitionsList	*	*	*	-	*
	wellFormedShadowCutIfBlockEntry	-	1	-	1	-
	BlocksRangeFromKernelStartIsBE	-	1	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	-	1	-	-	-
	sh1InChildLocationIsBE	-	1	1	-	-
	StructurePointerIsKS	1	1	-	-	-
	NextKSIsKS	-	1	-	-	1
	NextKSOffsetIsPADDR	-	1	-	-	1
	NoDupInFreeSlotsList	1	*	-	-	*
	freeSlotsListIsFreeSlot	1	1	1	1	1
	DisjointFreeSlotsLists	1	*	-	-	*
	inclFreeSlotsBlockEntries	1	1	-	-	1
	DisjointKSEntries	1	1	-	-	1
	noDupPartitionTree	*	*	-	-	*
	isParent	1	*	-	-	*
	isChild	1	*	-	-	*
	accessibleChildPaddrIsAccessibleIntoParent	*	*	-	-	*
	noDupKSEntriesList	1	*	-	-	*
	noDupMappedBlocksList	1	*	-	-	*
	noDupUsedPaddrList	1	*	-	-	*
	sharedBlockPointsToChild	*	1	*	-	*
	MPUInAccessibleBlocks	*	-	-	-	-
Security	verticalSharing	*	*	*	-	*
	partitionsIsolation	*	*	*	-	*
	kernelDataIsolation	*	*	*	-	*
	Total score	14+8*	15+13*	6+5*	3	7+14*

Table 12.2: Type changes impacts in Pip-MPU.

	Properties	pdentry.(firstfreeslot)	pdentry.(structure)	pdentry.(nbfreeslots)	pdentry.(parent)	pdentry.(MPU)
Consistency	nullAddrExists	-	-	-	-	-
	wellFormedFstShadowIfBlockEntry	-	-	-	-	-
	PDTIfPDFlag	-	-	-	-	-
	AccessibleNoPDFlag	-	-	-	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	1	-	-	-	-
	multiplexerIsPDT	-	-	-	-	-
	currentPartitionInPartitionsList	-	*	-	-	-
	wellFormedShadowCutIfBlockEntry	-	-	-	-	-
	BlocksRangeFromKernelStartIsBE	-	-	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	-	-	-	-	-
	sh1InChildLocationIsBE	-	-	-	-	-
	StructurePointerIsKS	-	1	-	-	-
	NextKSIsKS	-	-	-	-	-
	NextKSOffsetIsPADDR	-	-	-	-	-
	NoDupInFreeSlotsList	*	-	*	-	-
	freeSlotsListIsFreeSlot	*	-	*	-	-
	DisjointFreeSlotsLists	*	-	*	-	-
	inclFreeSlotsBlockEntries	1	1	1	-	-
	DisjointKSEntries	-	1	-	-	-
	noDupPartitionTree	-	*	-	-	-
	isParent	-	*	-	1	-
	isChild	-	*	-	1	-
	accessibleChildPaddrIsAccessibleIntoParent	-	*	-	-	-
	noDupKSEntriesList	-	*	-	-	-
	noDupMappedBlocksList	-	*	-	-	-
	noDupUsedPaddrList	-	*	-	-	-
	sharedBlockPointsToChild	-	*	-	-	-
	MPUInAccessibleBlocks	-	*	-	-	1
Security	verticalSharing	-	*	-	-	-
	partitionsIsolation	-	*	-	-	-
	kernelDataIsolation	-	*	-	-	-
	Total score	2+3*	3+12*	1+3*	2	1

Table 12.3: Field change impacts on Partition Descriptor entry fields in Pip-MPU.

	Properties	blockentry.(blockindex)	blockentry.(blockrange).(startAddr)	blockentry.(blockrange).(endAddr)	blockentry.(present)	blockentry.(accessible)	blockentry.(read)	blockentry.(write)	blockentry.(execute)
Consistency	nullAddrExists	-	-	-	-	-	-	-	-
	wellFormedFstShadowIfBlockEntry	-	-	-	-	-	-	-	-
	PDTIfPDFlag	-	1	-	1	1	-	-	-
	AccessibleNoPDFlag	-	-	-	-	1	-	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	-	1	-	1	1	1	1	1
	multiplexerIsPDT	-	-	-	-	-	-	-	-
	currentPartitionInPartitionsList	-	*	-	*	-	-	-	-
	wellFormedShadowCutIfBlockEntry	-	-	-	-	-	-	-	-
	BlocksRangeFromKernelStartIsBE	1	-	-	-	-	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	1	-	-	-	-	-	-	-
	sh1InChildLocationIsBE	-	-	-	-	-	-	-	-
	StructurePointerIsKS	1	-	-	-	-	-	-	-
	NextKSIsKS	1	-	-	-	-	-	-	-
	NextKSOffsetIsPADDR	1	-	-	-	-	-	-	-
	NoDupInFreeSlotsList	-	-	*	-	-	-	-	-
	freeSlotsListIsFreeSlot	-	1	*	1	1	1	1	1
	DisjointFreeSlotsLists	-	-	*	-	-	-	-	-
	inclFreeSlotsBlockEntries	-	-	1	-	-	-	-	-
	DisjointKSEntries	-	-	-	-	-	-	-	-
	noDupPartitionTree	-	-	-	1	-	-	-	-
	isParent	-	-	-	*	-	-	-	-
	isChild	-	-	-	*	-	-	-	-
	accessibleChildPaddrIsAccessibleIntoParent	-	*	*	*	*	-	-	-
	noDupKSEntriesList	-	-	-	-	-	-	-	-
	noDupMappedBlocksList	-	-	-	1	-	-	-	-
	noDupUsedPaddrList	-	*	*	*	-	-	-	-
	sharedBlockPointsToChild	-	*	*	*	-	-	-	-
	MPUInAccessibleBlocks	-	*	*	*	*	-	-	-
Security	verticalSharing	-	*	*	*	-	-	-	-
	partitionsIsolation	-	*	*	*	-	-	-	-
	kernelDataIsolation	-	*	*	*	*	-	-	-
	Total score	5	3+8*	1+10*	5+10*	4+3*	2	2	2

Table 12.4: Field change impacts on Block entry fields in Pip-MPU.

	Properties	sh1entry.(PDflag)	sh1entry.(PDchild)	sh1entry.(inChildLocation)	scentry.(origin)	scentry.(next)
Consistency	nullAddrExists	-	-	-	-	-
	wellFormedFstShadowIfBlockEntry	-	-	-	-	-
	PDTIfPDFlag	1	-	-	-	-
	AccessibleNoPDFlag	1	-	-	-	-
	FirstFreeSlotPointerIsBEAndFreeSlot	1	1	1	1	1
	multiplexerIsPDT	-	-	-	-	-
	currentPartitionInPartitionsList	*	-	-	-	-
	wellFormedShadowCutIfBlockEntry	-	-	-	-	-
	BlocksRangeFromKernelStartIsBE	-	-	-	-	-
	KernelStructureStartFromBlockEntryAddrIsKS	-	-	-	-	-
	sh1InChildLocationIsBE	-	-	1	-	-
	StructurePointerIsKS	-	-	-	-	-
	NextKSIsKS	-	-	-	-	-
	NextKSOffsetIsPADDR	-	-	-	-	-
	NoDupInFreeSlotsList	-	-	-	-	-
	freeSlotsListIsFreeSlot	1	1	1	1	1
	DisjointFreeSlotsLists	-	-	-	-	-
	inclFreeSlotsBlockEntries	-	-	-	-	-
	DisjointKSEntries	-	-	-	-	-
	noDupPartitionTree	-	-	-	-	-
	isParent	-	-	-	-	-
	isChild	-	-	-	-	-
	accessibleChildPaddrIsAccessibleIntoParent	-	-	-	-	-
	noDupKSEntriesList	-	-	-	-	-
	noDupMappedBlocksList	-	-	-	-	-
	noDupUsedPaddrList	-	-	-	-	-
	sharedBlockPointsToChild	1	1	-	-	-
	MPUInAccessibleBlocks	-	-	-	-	-
Security	verticalSharing	*	-	-	-	-
	partitionsIsolation	*	-	-	-	-
	kernelDataIsolation	*	-	-	-	-
	Total score	5 + 4*	3	3	2	2

Table 12.5: Field change impacts on Shadow 1 and Shadow Cut entry fields in Pip-MPU.

		Impact score on consistency properties	Impact score on security properties
Type	PDT type	14 + 5*	3*
	BE type	15 + 10*	3*
	SHE type	6 + 2*	3*
	SCE type	3	0
	PADDR type	7 + 11*	3*
Field	pdentry.(firstfreeslot)	2 + 3*	0
	pdentry.(structure)	3 + 9*	3*
	pdentry.(nbfreeslots)	1 + 3*	0
	pdentry.(parent)	2	0
	pdentry.(MPU)	1	0
	blockentry.(blockindex)	5	0
	blockentry.(blockrange).(startAddr)	3 + 5*	3*
	blockentry.(blockrange).(endAddr)	1 + 7*	3*
	blockentry.(present)	5 + 7*	3*
	blockentry.(accessible)	4 + 2*	1*
	blockentry.(read)	2	0
	blockentry.(write)	2	0
	blockentry.(execute)	2	0
	sh1entry.(PDflag)	5 + 1*	3*
	sh1entry.(PDchild)	3	0
	sh1entry.(inChildLocation)	3	0
	scentry.(origin)	2	0
	scentry.(next)	2	0

Table 12.6: Illustration of the type *TFPC* and field impact score matrix on the proof of Pip-MPU.

12.3.2 Proof impact score at the function/service level

At function level, the impact score directly depends on the instructions and their own impact score. Only the field impact score is considered for now. The type impact score is left aside for now because it is included in the field modification, however, will have a role in the proof complexity described next.

To analyse the impact score at function-level, or the proof element, we must read the previous Tables [12.2, 12.3, 12.4, 12.5] row-by-row instead of column-by-column. Indeed, as the proof targets are the properties, we must sum all the scores for all properties to prove for the set of modifications in the function.

Definition 12.3.1 (Function impact score)**.** For the set of properties P to prove ($|P| = p$), the instructions I of the function F ($|I| = n$) and the matrix relating the two sets $M = (P, I)$, the impact score IS of F is defined as

$$IS = \sum_{i=1,j=1}^{n,p} M[P_j, I_i]$$

Similarly, the impact score of a service is the sum of all scores of its proof elements. The properties to prove include all properties of the isolation invariant, including the security properties.

Definition 12.3.2 (Service impact score)**.** For a service S composed of the set of proof elements PE ($|PE| = pe$) which impact scores are registered in the set $ISPE$, and having instructions I ($|I| = n$) in its own main service function, the properties P of the isolation invariant ($|P| = p$), and considering the matrix relating the two sets $M = (P, I)$, then the impact score ISS of S is defined as

$$ISS_S = \sum_{i=1,j=1}^{n,p} M[P_j, I_i] + \sum_{i=1}^{pe} ISPE_i + C$$

$C = 1$ is an additional impact score added to each proof element because there are always local adjustments to apply.

We illustrate the metric with the `addMemoryBlock` service.

First, we compute the impact score of the inner function `insertNewEntry`. It has 14 instructions, of which 10 are modification instructions. By summing all impact scores of these modification instructions given Table 12.6, we obtain a total score of: (2+3*)+(1+3*)+(3+5*+3*)+(1+7*+3*)+(5+7*+3*)+(4+2*+1*)+2+2+2+2=24+37*.

Then, we compute the impact score of the main service function. It has 29 instructions, of which 2 modification instructions. The impact score of the main service

function is then 3+3=6.

Finally, we sum everything up and get the final impact score of 30+37*. This score is definitely higher than a service without any modification instructions, like `findBlock`, which has an impact score of 0.

12.3.3 Property complexity

The impact score is not sufficient to describe the difficulty to conduct the proofs. Indeed, the properties themselves participate in that difficulty.

I retain four property characteristics that elevate the property complexity: 1) the size of the property (the number of sub-propositions) 2) the number of lists 3) the number of variables involved 4) the number of final proof goals (the size of the last sub-propositions). Concerning the size of the property, it has a direct influence on the proof context, which must be adapted and transformed to solve the proof goal. About lists, we have seen that the proof impact could be lowered by setting up lemmas that state the expected list modifications. After the creation of the lemmas, only their number implies additional proof work. Regarding the number of variables, they imply additional combinations in the proof script. For example, if two partitions are expected, like in the security properties, the proof must explore the different combination of these partitions. Finally, *à propos* the number of final goals, they state the number of proofs to provide to solve the global proof goal.

Given these characteristics, we end up with the property complexity matrix represented in Table 12.7 computed for Pip-MPU's isolation invariant properties. The table shows heterogeneous complexities, with higher complexities in the security properties, and the properties *freeSlotsListIsFreeSlot*, *DisjointFreeSlotsLists*, *accessibleChildPaddrIsAccessibleIntoParent* and *sharedBlockPointsToChild*, which means more case discriminations in the proof script, more lists, richer proof context and more proof goals that complicate the proof.

12.3.4 Proof complexity

The proof complexity reflects the overall difficulty that requires problem-solving capabilities. It is a combination of the instruction impacts and the property complexities.

Definition 12.3.3 (Instruction proof complexity)**.** The proof complexity PC of an instruction I is defined as the multiplication of the corresponding column in the instruction impact matrix IIM with the total column in the property complexity matrix PCM.

$$PC_I = IIM[I] * PCM[Total]$$

	Properties	Size of property	Number of lists	Number of variables	Number of final proof goals	Total
Consistency	nullAddrExists	1	0	0	1	2
	wellFormedFstShadowIfBlockEntry	2	0	1	1	4
	PDTIfPDFlag	2	0	2	4	8
	AccessibleNoPDFlag	4	0	2	1	7
	FirstFreeSlotPointerIsBEAndFreeSlot	2	0	2	2	6
	multiplexerIsPDT	1	0	0	1	2
	currentPartitionInPartitionsList	1	1	0	1	3
	wellFormedShadowCutIfBlockEntry	2	0	1	2	5
	BlocksRangeFromKernelStartIsBE	3	0	2	1	6
	KernelStructureStartFromBlockEntryAddrIsKS	3	0	2	1	6
	sh1InChildLocationIsBE	3	0	2	1	6
	StructurePointerIsKS	2	0	2	1	5
	NextKSIsKS	5	0	3	1	9
	NextKSOffsetIsPADDR	3	0	2	1	6
	NoDupInFreeSlotsList	2	1	2	3	8
	freeSlotsListIsFreeSlot	6	1	4	1	12
	DisjointFreeSlotsLists	4	2	2	5	13
	inclFreeSlotsBlockEntries	3	2	1	1	7
	DisjointKSEntries	4	2	2	1	9
	noDupPartitionTree	1	1	1	1	4
	isParent	3	2	2	1	8
	isChild	3	2	2	1	8
	accessibleChildPaddrIsAccessibleIntoParent	4	4	3	1	12
	noDupKSEntriesList	2	1	1	1	5
	noDupMappedBlocksList	2	1	1	1	5
	noDupUsedPaddrList	2	1	1	1	5
	sharedBlockPointsToChild	7	5	5	2	19
	MPUInAccessibleBlocks	3	1	2	1	7
Security	verticalSharing	3	4	2	1	10
	partitionsIsolation	5	5	3	1	14
	kernelDataIsolation	3	4	2	1	10

Table 12.7: Property complexity matrix *PCM* in Pip-MPU.

At a function level, proof complexity indicates the difficulties brought by each instruction to prove the properties.

One particularity is the type of the entry which is modified. Indeed, types are included in the field change, in the sense that the type must be known to modify a value. However, from the proof point of view, the interest is not in the number of type modifications. Indeed, if several instructions modify the same address with the same type, we can extract the type property from the first instruction encountered and use it in all properties that require it.

Hence, the impact of type modification must be seen from a function perspective. It depends on which type is modified and how many different memory addresses are modified.

Definition 12.3.4 (Type function proof complexity)**.** Consider the map $M : I \rightarrow (T, A)$ mapping the instructions I to their type in the set of types T and the modified memory address from the set of memory addresses A.

The type function proof complexity $TFPC$ of function F is defined as the set of tuples $TA = \{(t, a) | t \in T \wedge a \in A\}$ of size s corresponding to M applied to all instructions of F multiplied with the corresponding type impact score TIS:

$$TFPC_F = \sum_{i=1}^{s} TIS[TA_i[t]]$$

For example, the function `insertNewEntry` manipulates three different types: PDT, BE and SCE. For each type, only one memory address is associated, so three different addresses. Hence, the part of the function proof complexity due to the types is $TIS[PDT] + TIS[BE] + TIS[SCE]$ =14 + 8* + 15 + 13* + 3 = 32 + 24*.

Finally, we can define the function proof complexity encompassing the instructions and the types used in the function.

Definition 12.3.5 (Function proof complexity)**.** The proof complexity PC of a function F composed of n instructions is defined as the sum of each instruction's proof complexity added to the type proof complexity of F.

$$PC_F = TFPC_F + \sum_{1}^{n} PC_i$$

For example, we consider the service `addMemoryBlock`.

The service's modification instructions in its main function are `writeSh1PDChildFrom BlockEntryAddr`, which modifies the PDchild field of the Shadow 1 entry, and `writeSh1 InChildLocationFromBlockEntryAddr`, which modifies the InChildLocation field of

Instruction proof complexity	
Instructions	Proof complexity
writePDFirstFreeSlotPointer	13 + 3*
writePDNbFreeSlots	7 + 3*
writeBlockStartFromBlockEntryAddr	26 + 8*
writeBlockEndFromBlockEntryAddr	7 + 10*
writeBlockAccessibleFromBlockEntryAddr	33 + 3*
writeBlockPresentFromBlockEntryAddr	35 +10*
writeBlockRFromBlockEntryAddr	18
writeBlockWFromBlockEntryAddr	18
writeBlockXFromBlockEntryAddr	18
writeSCOriginFromBlockEntryAddr	18
Sub-total	193 + 37*
Type function proof complexity	
Number of different PDT type addresses : 1	14+8*
Number of different BE type addresses : 1	15+13*
Number of different SHE type addresses : 0	0
Number of different SCE type addresses : 1	3
Number of different PADDR type addresses : 0	0
Sub-total	35 + 21*
Total	228 + 58*

Table 12.8: Illustration of the computation of `insertNewEntry`'s proof complexity.

the same Shadow 1 entry. They have respectively a proof complexity of $PC_{sh1entry.pdchild} = 6*1+12*1+19*1 = 37$ and $PC_{sh1entry.inchildlication} = 6*1+6*1+12*1 = 24$.

The function `insertNewEntry` has a proof complexity of $PC_{insertNewEntry} = 228+58*$ as computed in Table 12.8 from IIMs [12.2, 12.3, 12.4, 12.5], $TFPC$ 12.6 and PCM 12.7, if we consider all consistency properties. `addMemoryBlock`'s main function proof complexity is then the sum of the two computed proof complexities : $PC_{addMemoryBlock} = 37+24+228+58* = 289+58*$.

At a system level, the global proof complexity depends on the proof elements that are the most difficult to prove.

Definition 12.3.6 (Overall proof complexity)**.** The overall proof complexity OPC is the highest proof complexity in the set of proof complexities PC found in the service-level proofs:

$$OPC = max(PC)$$

12.3.5 Proof effort

The proof complexity must be distinguished from the proof effort 12.1.2.

There are many ways to compute the proof effort. For example, it can be retrospectively computed as the person-hours or person-months dedicated to the formal verification activity. Unfortunately, this measure does not consider the difficulty of the task and the experience of the proof developers. As such, man-months is an excellent standard measurement of the amount of work and can be used for financial tracking, however, little does it say if the same work could be done faster and how difficult it was.

In this work, we introduce the notion of estimated proof effort.

Definition 12.3.7 (Estimated proof effort)**.** The estimated proof effort EPE relies on the proof complexity PC of the n ($n \in \mathbb{N}$) proof elements to prove.

It is defined as:

$$EPE = n + \sum_{i=1}^{n} PC_i$$

n again represent the one-time cost to locally adapt the proofs to the reusable elements.

With EPE, one can plan the work in advance and has an idea of the complexity of the task. EPE can be adapted to any project given a clear definition of the expected proof complexity.

12.4 Proof path best effort strategy

In the previous chapter, we adopted a blind proof approach (horizontal and vertical explorations) to cover all services and rapidly discover the properties to prove.

`addMemoryBlock` had been chosen for the first vertical exploration because it touched many memory types, had an internal function, modified the state, and touched many properties of which security properties. It allowed us to test our proof process. Retrospectively, with the matrix from the previous section, the service has a proof complexity score of 289 + 53*. The score is high, but not as high as `cutMemoryBlock` because the latter not only relies on the same internal function but also on additional functions.

Now that formalisation is set up and properties discovered, is it possible to optimise the remaining elements to prove? In other words, the overall proof effort can be estimated, but is there any order in the proof elements that is better than another? In the following, we discuss the proof path strategy followed in Pip-MPU that proves incrementally the services by minimising the proof effort at each increment.

12.4.1 Proof path strategy

The formal verification process is only finished once all proof elements are proven. However, there are multiple ways to reach the ultimate goal; for example by proving the services by alphabetical order or by proving first all internal functions and primitives and later the services.

In Pip-MPU, the chosen strategy is to provide a proof coverage metric by validating the services one by one. It seems more reliable to be able to prove a service first, than all internal functions, because it challenges the full isolation invariant, including the security properties, and it allows a certain room for change in the design of unproved elements (otherwise it implies changes in existing proofs). The order in which the services must be proved is computed thanks to the Algorithm 3 where the goal is to minimise the current proof effort and to maximise the proof coverage. Indeed, we favour a service which proof effort is lower than others, but also permits to lower the proof effort of the remaining services because its proof elements are reused. Hence, a high reusability metric 12.2.8 provides more flexibility to chose the best proof path strategy.

Algorithm 3 Proof path best effort strategy algorithm.

```
1: service_order ← []
2: current_proof_effort ← []
3: next_proof_effort ← []
4: repeat
5:    for service in services do
6:       proof_path ← get_proof_path(service) { Get the proof path for service}
7:       current_proof_effort[service] ← get_current_proof_effort(proof_path) { Com-
         pute the current proof effort }
8:       next_proof_effort[service] ← get_next_proof_effort(proof_path) { Compute
         the benefit for the other services }
9:    end for
10:   current_proof_effort_min ← get_services_with_min_current_proof_effort (cur-
      rent_proof_effort) {Take the minimum of the first value}
11:   if length(current_proof_effort_min) > 1 {if several services have the same cur-
      rent proof effort} then
12:      next_proof_effort_max ← get_services_with_max_next_proof_effort (cur-
         rent_proof_effort_min, next_proof_effort) {Take the maximum of the second
         value out of the services with the minimum current effort}
13:      selected_service ← next_proof_effort_max[1] {Always take the first service in
         the list, even if several services have the same score }
14:   else
15:      selected_service ← current_proof_effort_min[1]
16:   end if
17:   service_order.add(selected_service) {Add the selected service in the service order
      list}
18:   remove_proof_path(selected_service) { Remove all elements of the chosen ser-
      vice's proof path }
19: until all services in service_order
20: return service_order{ Returns the service order}
```

Iteration	addMemoryBlock	cutMemoryBlock	mapMPU	mergeMemoryBlocks	prepare	createPartition	collect	deletePartition	removeMemoryBlock	readMPU	findBlock
1	2/1	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	1/0°	1/0
2	2/1	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	0/0	1/0°
3	2/1°	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	-	0/0
4	0/0	6/10	4/3°	9/10	12/8	8/9	7/7	7/6	6/4	-	-
5	-	4/7°	0/0	6/1	12/8	8/9	7/7	7/6	6/4	-	-
6	-	0/0	-	4/3°	7/3	6/4	4/0	5/1	4/0	-	-
7	-	-	-	0/0	5/0	4/1°	4/0	5/1	4/0	-	-
8	-	-	-	-	5/0	0/0	4/0	3/0°	4/0	-	-
9	-	-	-	-	5/0	-	4/0°	0/0	4/0	-	-
10	-	-	-	-	5/0	-	0/0	-	4/0°	-	-
11	-	-	-	-	5/0°	-	-	-	0/0	-	-
12	-	-	-	-	0/0	-	-	-	-	-	-

Table 12.9: Proof path best effort strategy iterations of Algorithm 3 applied to Pip-MPU. Each iteration reveals a new score depending on previously selected services. The score is given as a tuple: $current\_proof\_effort/next\_proof\_effort$. A service is marked with '°' at the iteration when it is selected. Final service order: [readMPU, findBlock, addMemoryBlock, mapMPU, cutMemoryBlock, mergeMemoryBlocks, createPartition, deletePartition, collect, removeMemoryBlock, prepare]

12.4.2 Illustration of Pip-MPU's proof path strategy

We illustrate the Algorithm 3 with Pip-MPU's proof path strategy in Table 12.9.

For the sake of simplicity, we approximate the proof effort computations. The current proof effort on a proof path is the number of proof elements on the path. The next proof effort is computed as a sum of all incoming edges on a proof element minus its own incoming edge on the proof path (only keeping the additional incoming edges). Indeed, as described earlier, incoming edges on a proof dependency path represent the reusable lemmas which benefit other services as well.

At each iteration, the current proof effort decreases for the other services only if they had reusable lemmas also required in the last selected service. Note that this illustration is an example that can be deepened with more accurate proof effort based on previously presented estimate proof effort EPE. In such case, for instance at iteration 10, we would have probably chosen `removeMemoryBlocks` instead of `collect` because the proof complexity analysis would have shown a lower score.

12.5 Proof dashboard

The proof dashboard is a dynamic tool that reports the proof status. It modifies the perspective on proofs as proof development progresses. The dynamism is required given our proof strategy, with horizontal and vertical explorations and to adjust with incoming events like new discovered consistency properties or design modifications, as well as to report the progress of a long-lasting activity.

At the beginning of Pip-MPU's proof development, we have the complete view of the proof dependency graph. The goal is to develop all the proof elements. We can then present the proof status by a graph coverage.

However, having a coverage at the proof-element-level is not precise enough and the proof status would not change for a long time. For example, with `addMemoryBlock` it would mean to wait for the proof of `insertNewEntry` which takes a lot of proof effort. In addition to that, the detailed proof complexities presented in Section 12.3 cannot be computed because consistency properties are unknown at that stage. Thus, we would like an instruction-level rough estimation for the first moments of the proof development.

The first estimate is based on the number of instructions which have been visited during our Hoare instruction breakdowns. The current proof pointer gives the ratio of proof coverage depending on the number of remaining instructions in a service. In the horizontal exploration phase, this gives an advancement report, with underlying formation of the formalisation and the checking primitives. The Figure 12.2 illustrates the proof coverage after the horizontal exploration stage.

The remaining proofs are actually composed of new proof elements, such as functions and write instructions, but also embed similar instructions to what has already been proven, like read instructions. In such case, the proof is facilitated, only requiring local adjustments. Figure 12.2 also shows the amount of reusable elements already proven, which should not be difficult to pass.

However, concerning proof elements that have not yet been similarly proven, there are uncertainties. The intuition guides us to state that the remaining proofs are not of equal complexity if we just consider the instruction: a write instruction might be hard, but harder to prove is a function with several write instructions. As such, we approach the proof complexity by knowing four function characteristics that give a minimal sense of the proof effort: 1) the number of instructions 2) the instruction modifies the memory state 3) the instruction calls an inner function 4) the function is recursive because a loop invariant must be found. The proof dashboard then shows, in the remaining proofs, what amount is really challenging.

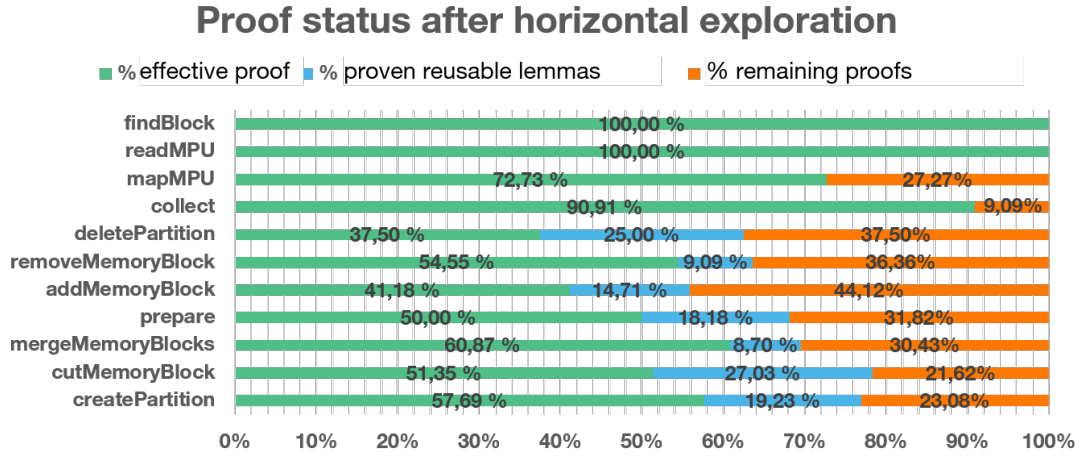


Figure 12.2: Proof status after the horizontal exploration stage approximated by proof elements, not considering the proof complexities.

Granularity	Metric
All services	Proof coverage by number of proven code instructions
For a function	Proof complexity (rough estimation or fined-grain analysis)
For a service	Number of proven properties of the isolation invariant

Table 12.10: Proof dashboard overview: metrics depending on the granularity.

During the vertical exploration stage, we zoom on a service. In addition to the current proof pointer and the amount of difficult proofs, we are now able to make a fine-grained analysis of the proof complexity as described in Section 12.3. The service's proof complexity score becomes the new goal and it is possible to compute the estimated proof effort, which can be transformed in time and tasks. The dashboard then reports the remaining proof activities to complete the proof of the service until the last instruction.

At the last instruction, the proof goal is the isolation invariant. The dashboard then breaks the latter property by property. The amount of proven properties compared to the complete set of properties reports the advancement.

12.6 Discussion

The world of metrics is large and the selection of meaningful metrics is hard.

For example, in Pip-MPU, because the kernel is not fully formally verified yet, we could also add the metric of the keyword *Admitted* which replaces the keyword *Qed* when all proof obligations have not been satisfied. But we might want a more fined-grain analysis, so not just the fact that it is partially proven, but *how* much the code is

really proven. However, this raises the issue of the proof complexity, that we provide in Section 12.3, that can be further quantified.

The same goes for the ratio of the number of SLOC over the number of proof tactics: give a sense of the higher effort to develop the proofs than to develop the code of the corresponding service. But this does not consider the design time of the services which comes before the actual development of the services. This leads to the involvement of the design time in the proof process comparison; for example the ratio of the design time with the service development time against the formalisation time and the proof of all services.

The granularity which gives the sufficient sense of understanding and the minimal set of meaningful metrics are not fixed yet.

More than that, common metrics in different verification projects might not be computed the same and comparisons might end in false interpretations.

For example, the definition of lines of code and lines of proofs are different: does it include comments, empty spaces, function prototypes? Also, different interactive theorem provers are used across these projects, so the measurement of lines of proof means different things. It also depends on the proof style of the proof developer or the development team. In Pip-MPU, the clarity and understanding of proofs implies many comments, proof scripts decomposition, sometimes duplicates, that penalises a metric like the ratio of code and proof lines. This is why comments are removed from the metric because they do not essentially mean a higher effort in proofs.

Furthermore, the proof complexity scores can be further deepened to be as close as possible to a realistic expected proof effort.

To take an example, in any of our scores, we only considered additional instructions as elevating the complexity. This might not always be true. As an illustration, the simple case where an instruction writes another value at some address and just after writes back the initial value at the same address again, does not modify the state at the end, however disturbs the proofs and properties. Our complexity score also omits any proof complexities on the lists because we said we created specific lemmas to apply at the modification instruction, which reduces the complexity. However, this is without considering that some lemmas relies on consistency properties, for example the list `getKSEntries` (get the list of slots in a partition) relies on $StructurePointerIsKS$ in case the modification affects a PDT type, because any change to the *structure* field of the modified PD entry completely changes the list. The cost of the lemma is then non-trivial.

Proof and code metrics must be further linked together to capture these combinations.

In addition to that, we advocate for a high code modularity that can be inherited by the proofs. This is usually not expected from a pure software perspective, where it has significant advantages like ease of code maintenance, however implies more generalisation costs by a longer design time and refactoring.

Without modularity, there might be less time in the design process which would facilitate the proofs by copy-pasting similar proof scripts. However, similar patterns would be difficult to track and a lot of time would be lost during maintenance.

Also, a proof development largely based on proof modularity includes a pre-analysis of the proof elements. It helps to reduce the mistakes of proving several times a similar lemma by knowing in which other service it has been already proven.

A high reusability score entails plenty of proof bricks to assemble for further development, for example if the API is extending to new services.

Finally, the impact scores give an idea of impacts on the current set of properties, which is not yet final. While the horizontal exploration identified most of the consistency properties of the system, we expect some modifications of the set, and in turn the scores. For example, the table currently gives us the impression that the Shadow Cut fields do not impact the properties very much, which is true for the current set of properties. However, we expect some properties about these fields to be discovered for the services `cutMemoryBlock` and `mergeMemoryBlocks`, which will change the score matrix.

12.7 Conclusion

Metrics enable to qualitatively or quantitatively measure some characteristics that help the proof process during design, verification and maintenance stages.

We propose here new metrics to track and analyse the proof process. We present the effective reusability metric that demonstrates the benefits of matching proof elements with code modularity. We introduce proof complexity scores which depend on the impacts on the isolation invariant and is computed from the instruction-to-the service perspective. This enables a definition of the estimated proof effort that can be used to split the work and track the achievements.

Furthermore, our proposal encompasses a procedure to select the best proof path strategy. The goal is to make an informed choice of a proof direction with optimisations. In Pip-MPU, the proof increment is the service. The order in which services are proven is chosen to minimise the current proof effort and maximise the benefits for the other proof elements. The proof effort progressively incorporates weighted proof complexities in order to assess the real effort.

Moreover, we construct a proof dashboard that dynamically reports the proof progress. The current proof view is refined by proposing different metrics for vari-

ous proof stages (rough estimates of proof coverage or detailed analysis with complexity scores).

Review of Part III

Chapter outline

Review

Pip-MPU exhibits strong isolation guarantees by the use of formal methods. The formal verification demonstrates that Pip's security properties, that ensure spatial partitioning, are satisfied by Pip-MPU. Several Pip-MPU services have been formally verified for isolation.

Pip-MPU relies on Pip (MMU)'s proof workflow and could reuse low-level proofs on memory manipulation. However, Pip (MMU) is too strongly rooted in the concept of memory page due to the use of the MMU and virtual memory. This results in totally different code bases, memory state and properties expressions, and thus totally different high-level proofs. Nevertheless, it demonstrates the successful adaptation of Pip (MMU)'s proof framework performed at implementation level.

The formal verification of the services also questioned the process and the proof path. Pip-MPU did not follow the co-design process used in Pip (MMU) but rather leverages novel techniques based on proof checkpoints. This work also modified the memory types that revealed efficient ways to discriminate cases and simplified the proofs by requiring a smaller proof context. Furthermore, this part unveiled proof strategies to efficiently discover the consistency properties which depend on the memory representation and hence different from Pip (MMU) but with some similarities. The horizontal and vertical exploration strategies developed in this work enable a fast covering of many consistency properties that facilitate the proofs of the first modification service and proofs thereafter. Indeed, each time a new consistency property is discovered, it must also be satisfied

in all other proofs already developed which severely impacts the proof effort. With a good coverage from start, there will be less iterations on the proofs and less global proof efforts.

The formal verification process followed a best effort proof path. Specific proof metrics have been developed to reveal the most efficient proof path that increases the number of proven services while having a minimal proof effort for the next service to prove. Several subsidiary metrics have been developed to refine the analysis on the proof effort of software components in the form of a dynamic dashboard reporting the progress depending on the proof complexity. These metrics rely on the intrinsic expressions of the properties and the dependency graph linking the services based on the shared internal functions.

Perspectives

Pip-MPU and Pip (MMU) do not share the same hardware platform. However, Pip-MPU's verification process demonstrates that proofs can be adapted even if the hardware base changes. Pip's partitioning scheme could be adapted for other platforms, for example using the SAU (Security Attribution Unit) that is close to the MPU but present only for ARMv8 versions or even memory capabilities (much like what is currently done on the Morello board [10]) which would challenge the design and the verification process even more.

Furthermore, the proof metrics developed in this part are useful information to select a verification path. Because the proof development cost easily overtakes the design cost, it is necessary to find new metrics and proof strategies that lower the overall proof effort.

In addition to that, proofs are meant to last, especially given the verification cost and the strong guarantees it demonstrates. Research is also investigating the maintenance of proofs, for example proof repair tools [150], to keep the proofs up-to-date when the code base changes.

At last, Pip-MPU's proof assumptions are similar to Pip (MMU)'s, other formally verified projects and Hoare logic limitations [90]. The goal is to lower them down which erases the minimal trust required to perform the verification, *e.g* in tools, toolchains, and hardware platforms. With Pip-MPU, we already have the possibility to use a trusted toolchain from Coq to the machine code, by leveraging the δx tool and CompCert [186], toolchain that has already been successfully utilised [188]. Nevertheless, two links are missing to have a complete end-to-end verification from the design to the hardware: the hardware itself and the HAL manually written in C. About the hardware, formal hardware specification have recently being investigated [147] for different processor architectures. A machine-readable specification could then be connected to the hardware

model used in Pip. Concerning the HAL, we could eventually use CompCert to construct the HAL's AST and import it into Coq. There, it should connect or replace the current HAL model.

Takeaways

- We formally verify that Pip-MPU follows Pip's security properties, *i.e.* Pip-MPU replicates Pip's partition tree (notably isolation)
- We have a partial proof of the isolation property in Pip-MPU's services (two services fully formally verified, one almost complete), both informal and formal by the use of the Coq proof assistant
- The proofs are conducted at implementation level
- We inherit Pip's proof workflow with evolution on the proof techniques
- We develop novel proof metrics to drive the proof path with minimal stepwise efforts
- We propose to track the proof progress by a dynamic proof dashboard that adapts the metrics for various proof stages

Conclusion

Chapter outline

Contributions and perspectives

This work compiled the following main contributions:

- A framework to set up MPU-protected software components in a flexible and hierarchical design
- Pip-MPU, the Pip protokernel adaptation for constrained devices
- The formal verification of Pip-MPU's security properties

- Proof techniques and proof metrics to drive the formal verification process and communicate the proof progress to non-experts

All in all, this work demonstrated the design, implementation and verification processes of Pip-MPU, a secure kernel for constrained devices, by adapting the proposed framework to Pip's strict security policy and by formally verifying Pip's security properties by minimising the proof efforts. Pip-MPU not only thwarts cyber attacks of our considered attacker model (*cf.* Section 7.7) but also do so with a high-level of confidence.

The MPU-based memory isolation solution presented in this thesis outperforms current solutions by being way more flexible.

Beyond the formally verified Pip-MPU kernel, this work showed broader applications of the contributions that echo to our thesis announced in Section 4.1.

A framework to set up MPU-protected software components in a flexible and hierarchical design The framework is generic and portable. Operating systems and applications with different global security policies than Pip's can customise the framework and still benefit from the flexibility for complex applications, finer-grained control over the system components and many implicit privilege levels, while ensuring MPU-protected domains. Furthermore, the framework also allows custom local policies, within the frame of the global security policy, which further adjust the security model at closest to the needs. Protection from stack smashing or from the network stack, $W \hat{} X$ principle enforcement, isolation of the OS from the applications and between applications, are some examples of possible local security policies that can be set up by the framework.

Pip-MPU, the Pip protokernel adaptation for constrained devices This dissertation describes the successful adaptation of an MMU-based separation kernel to be MPU-based. Despite the hard constraints on resources, we demonstrated the flexibility as well as security could be replicated.

Pip-MPU is an isolation layer that can be deployed on most constrained devices. The evaluation showed great security benefits for reasonable downgrades in performance and energy. With my research teams, we are currently successful to port the RIOT OS [151] over Pip-MPU. These are parallel activities to this work, however validate the position of Pip-MPU as a secure-by-design basis for existing bare-metal applications and multi-threaded operating systems in real-world scenarios. The port comes without functional loss.

The formal verification of Pip-MPU's security properties Confidence in an MPU-based isolation kernel has never been so high in other systems than with Pip-MPU's

security properties formal proofs.

The glitch from an MMU-based to an MPU-based hardware platform showed equivalent guarantees, provided deep adaptation because of divergent hardware models. This work stressed the impacts of this glitch and also presented means to harmonise the formalisation to increase proof reusability.

In addition to that, proofs give insights on the assumptions required to obtain the desired properties, even hardware assumptions. With Hoare logic, changing hardware platform changes the initial assertions, which must imply the current pre-conditions to keep the program's correctness, otherwise the proofs fail. It explicitly shows what is unconsciously inferred by the developer and would have shown in Pip-MPU's case the impossibility to reuse the same code because the specification (the expressions of the security properties based on an incompatible hardware model) do not match without adaptation. The developer is not tricked into using a program in an inconsistent way that would invalidate the proofs, and so avoids unpleasant surprises when executing the code.

Furthermore, the Pip development team currently studies the propagation of the proven properties to the application-level.

Proof techniques and proof metrics to drive the formal verification process Proof development takes time and intellectual efforts, much more than the implementation and the design phases. Applying the proof strategies presented in this dissertation, at local (service) and global (full API) levels, can reduce them. The proof techniques accelerate the discovery of important properties for the final proof goal. They can be leveraged for other formal verification projects and give alternatives when other techniques do not fit some specific proof stage like co-design of code and proofs which implies returns to the design phase.

We also discussed new proof metrics that compare different approaches and give a finer understanding of the properties to prove to select the best proof path. Proof metrics are in its infancy and much is to be done in this area to comprehend the proof development process from a proof engineering point of view. Proof efforts are still high, and this work shows no exception; however so does testing as well as bug finding and correction during the lifetime of a product. Proofs showing less effort could be more cost-efficient by identifying all these bugs and undefined behaviours prior to execution and market launch.

Open-source All the contributions developed in this work are publicly released under open-source licences. This is the foundation for a widespread adoption of the solution or an open inspiration for systems that want to exhibit strong security guarantees to

participate in a global security outreach effort in the ecosystem of constrained devices, but also to get higher market value at lower cost.

Limitations of the approach

One rigid limitation is of course the availability of the MPU, which is an optional unit. Many reasons let developers put aside the MPU even when there is one, should it be because of the hardware constraints, the power it drains, or the time-to-market pressure and lack of time to set it up. However, we argue the protection it provides is sufficient to reach a high level of security when correctly configured and Pip-MPU is formally verified for that.

Furthermore, some changes are expected on existing MPU-based software components or bare-metal applications to use Pip-MPU. Our current work-in-progress RIOT OS port shows no great difficulties, however non trivial changes could be required for other systems that want to benefit from the flexibility and determinism of some of the system calls like systems with real-time constraints that can only be assessed by doing a port.

Proof assumptions are minimal and similar to other formally verified kernels. One of the most unsure assumptions is that the hardware correctly behaves. No formal specification, and no formal verification of the hardware implementation are currently at disposal. This work is based on a hardware specification extracted from our understanding of the inner workings of the hardware components. It is a simplified view that does not capture all implementation specificities. Recent research unfortunately showed this assumption fails for current hardware components, like Spectre and Meltdown [110, 120]. Fortunately, processor designers now aim to fill the security gap by evolving processor architectures tampered against micro-architectural attacks, like the given example of Arm and its Morello board. On a side note, this timing shows we enter a mature functional era where cyber security is really being investigated at any level, from hardware to software.

The formal verification process of Pip-MPU is not finished and most of the services remain to be proven. The most difficult part of the verification journey has nevertheless been reached, with a fully functional formal baseline and many developed theorems that greatly facilitate the remaining parts to prove. The limitation is of course the size of the elements to prove. Pip-MPU has a small code base which reduces the attack surface but especially reduces the number of elements to fully prove. Small systems like embedded systems are simple enough to apply formal verification in real-world scenarios.

Despite these encouraging first steps, formal verification raises supplementary difficulties. The proof development is a long-lasting effort, but also requires expertise, not

even evident for "programmers of high caliber"(Tony Hoare) [90]. Finding such human resources are difficult, especially stemming from computer engineering grad students, and the necessary time to train them is as much time less spent on research.

Insights and learnings on the design of Pip-MPU

The creation of Pip-MPU, by adapting the existing Pip protokernel, was not a straightforward journey. Overcoming this challenge meant solving several non-trivial issues by making informed choices on the design and the methodology. I wish to give here some insights of our thoughts and the critical choices we made for future adaptations of Pip or other systems also required to change their hypotheses on the underlying hardware. The following sections are partially based on work initially discussed in the paper [53]. This section is voluntarily informal and topics are approached in more general terms.

History and methodology

Trade-off is usually the golden word in embedded systems, especially constrained devices, when it comes to satisfying their requirements. The low-end embedded system's expectations, notably composed of the SWaP-C factors, are evolving to reach another level of complexity, for a more agile and secure ecosystem (for example IETF SUIT [92] for software updates for IoT published last year (2021)).

The notions described in the previous chapters do not follow the chronological order. Indeed, from the very start, the adaptation of Pip for constrained devices was the target, as the flexibility and security brought by Pip would benefit the IoT ecosystem as explained above. However, we found no example of a secure MMU-based systems that had been adapted to MPU-based platforms. Our initial leads (*cf.* section III below) lifted more challenges than answers.

We then decided to conduct a survey of state-of-the art MPU-based memory isolation solutions by exploring the available documentation and source codes. Section III indicates, with extended remarks of previous state-of-the-art comments from Chapter 3, an active community proposing flourishing ideas to overcome some of the challenges. However, none really fit our problematic, and as Pip requires both flexibility and security, many times presented as incompatible aspects in the state-of-the-art solutions, we were back to our initial point. In the opposite, Pip's genericity was compatible with almost all, if not all, solutions as a primary basis for their memory isolation.

We elaborated Pip-MPU's design phases as multiple feedback loops illustrated in Figure 12.4. This activity led to multiple scenarios setting up Pip's partitioning scheme. Anticipated impacts on performances, algorithm complexity, memory, flexibility, proofs were closely monitored that revealed candidate scenarios. Some scenarios shared char-

acteristics, so we identified the ones that were the most adapted to our requirements as developed below. Furthermore, as Pip-MPU is an adaptation of Pip (MMU), they were initially expected to be similar, so we connected all scenarios based on the identified characteristics that disclosed the efforts of adaptation, starting from Pip (MMU)'s design. We selected the path reducing the efforts while reducing the impacts at most (again, trade-offs had to be made, especially for technical details coupled with the MPU as discussed in the previous chapters).

The scenarios pointed out that other systems might have lighter or stronger requirements than Pip-MPU, which tries to be the most generic as possible. Especially concerning the security policy, our preliminary studies showed lots of disparities in how they use the MPU for the targeted protection. The framework (Chapter 6) was born in this context, a generalisation of Pip's concepts. It gives more flexibility to current systems and allows them to customise their systems policies to match their own requirements and use cases.

Concerning the development of the retained scenario, we used a step-by-step approach. We first developed pseudo-codes for our services, implemented them in Python scripts with unit tests, and then transposed them in Gallina within Pip's development pipeline. While we forked Pip's project for our initial intervention on the code, we ended up to change it almost completely because the original code was too tightly coupled with the notion of a memory page, which has no more sense for Pip-MPU (at the same time too rigid with fixed size pages, but also too flexible with millions of available pages at system start).

I explain in the following the different points I mentioned above. They testify our delicate trials to select the complex combination of performance, security, and features which composes Pip-MPU.

Initial thoughts and technological locks

High-end and low-end devices belong to different classes of devices featuring different hardware platforms. We quickly identified that the MMU was not available on low-end devices and that the access control role could be overtaken by the MPU. The MPU seemed to be a good candidate given the MPU region protection with configurable access permission rights. The MPU solved the security aspect and would be our hardware base, the design of Pip-MPU started. But we rapidly faced the issue of flexibility.

Indeed, we first imagined a one-to-one relation between the MPU region and the MMU memory page. This would have been possible by removing all indirection levels of the MMU, as if we ended up with the last MMU level only with a reduced size. However, the limitation was too strong with only an 8 (maximum 16) entry-table for the last MMU level corresponding to the number of MPU regions. There was no

flexibility, or immensely reduced if we considered two MPU regions per partition (code and data regions), and peripherals were out of consideration. Hence, we formulated the requirements for Pip-MPU: same behaviour as Pip (MMU) and so very flexible, same guarantees of isolation and thus formally verified, adapted to constrained objects (memory footprint, reasonable complexity depending on the operation, low impacts on performances and energy consumption). As formal verification of Pip-MPU's services was required, we already targeted to keep as many Pip structures as possible because they are tied to the underlying proofs. Our initial hope was to reuse many parts of Pip's proofs if we kept close to the structures (which ended up to be false, see next part on formal verification for more insights on this matter).

Our search for flexibility led us first to conduct a survey on existing solutions using the MPU to set up some form of memory isolation.

Preliminary studies

This section does not aim to repeat previous sections of the dissertation, but to propose another reading grid.

A flourishing state of the art... Chapter 3 showed the existence of many systems that offer memory isolation for constrained devices (mostly ARM Cortex-M and many of them are open-source). We know by their sole existence and date of release that research on this topic has accelerated in the last ten years. We also understand there is a search for security and flexibility/agility, for example with µVisor and its sandboxes (developed by ARM, so they also tried to find solutions for the technical hassle of the MPU region alignment, which maybe led to the simplified ARMv8-MPU), TockOS by loading/unloading applications on the fly, the lost effort of security by mixing ACES compartments because of hardware limitations that OPEC proposes to solve. We also see the industry, governmental agencies and the society in general showing more interest in cyber security, with Arm and its Morello board [17] integrating hardware capabilities as instruction extensions (unfortunately only available for the Cortex-A family at first, the Cortex-M had already been second to receive updates with the adapted ARMv8 architecture), the French-German project TinyPART [174] which involves Pip-MPU, the French National Agency for the Security of Information Systems (ANSSI) that proposed EwoK [25] we described earlier, the French cyber warfare unit that was created in 2017 [128] or the French Cyber Campus created in 2022 [31] (this dissertation is written in 2022), and even the media and the population speak more about it as illustrated in Figure 12.3.

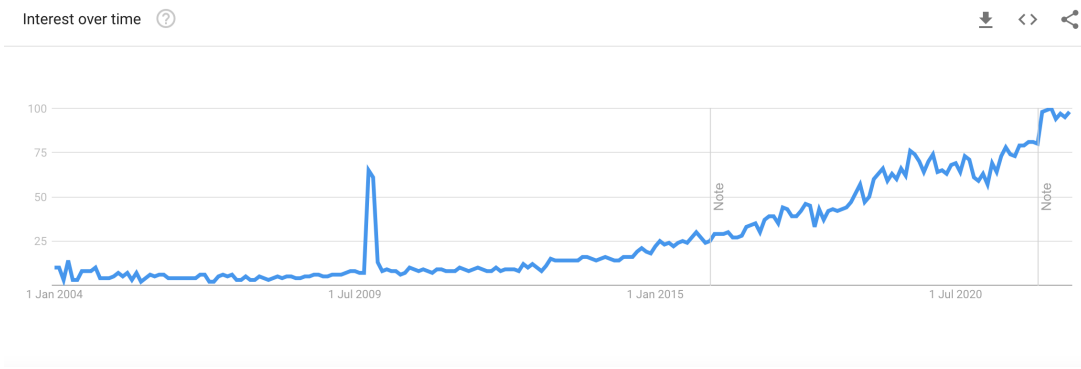


Figure 12.3: Screenshot from the results of the keyword "cyber security" from Google Trends [80] (27/08/2022)

...in contrast with real-world projects... However, there is a strong disparity between the mentioned efforts and the reality of the market. Time-to-market drives the embedded development and only 4% of the design time of an embedded system is spent on security/privacy threat/risk assessment [61].

We should also acknowledge the shift to connectivity: from legacy embedded systems to IoT devices. According to the 2019 Embedded Markets Study [61], 65% of the respondents linked to the embedded world were to have one or more projects devoted to IoT (compared to 21% at survey time). According to the Eclipse IoT Developer Survey 2019 [71], two-thirds of the respondents were currently working on IoT projects or were planning in the next 18 months. This clearly shows a movement towards connected devices. Overall, on the 13-30 billions of IoT devices expected on the market in 2022 [168, 76], even forecasted (or awaited) to be trillion in 2025 [166], the great majority of them will suffer from cyber attacks and the new business ecosystem will be the playground of cyber crimes capturing 300 billion to 2 trillion dollars worldwide and every year [146].

The shift towards connectivity seems to reflect the current state of security in IoT projects, as security is the top IoT developer concern (ranked before connectivity, data collection and analytics, and performance [71]). And, quite expected, non-IoT developers consider security is the top 3 greatest technology challenge [61], clearly showing the increasing need for security. Indeed, the interconnection between IoT and traditional IT systems gives more opportunities for malicious IoT devices to have a larger impact than non-connected devices. With potentially high impacts, the ecosystem is in a stringent need of high-level guarantees.

While systems receive more security attention now than ever before, no solution that we studied was suitable enough to the dynamic ecosystem of the IoT.

...which deals with a strong human factor (IoT developers' expectations) New technology adoption is not straightforward in the embedded world, and system designers have their expectations [61]. It comes out that most of the designers rely on a 32-bit architecture, with ARM architectures prevailing in this category, and they usually use the same processor across their projects. For the operating system, they use less and less in-house/custom OSes. If they have an OS, they are happy with the current solutions and have no reason to switch, also to be able to maintain software compatibility and make use of the expertise and familiarity. The reasons for changing operating system are mostly because the hardware or processor changed or because the OS was imposed without any involvement from the system designers. It shows they are not willing to change their development environment without any major factor, mostly independent of their will. Among the most important factors to choose an OS is the full access to the source code, the availability of the technical support, the compatibility with other software, systems and tools, no royalties and real-time performances.

Overall, it feels like the embedded designers are happy with the current state but will be unable to catch the IoT shift in a smooth manner. Billions of devices will enter the market, unprotected and unaware of the threats of the connected world. Given the depicted scenario, the ecosystem is even surprisingly calm. But this builds a time bomb, stretching over years, and industries might suffer from several waves of massive IoT attacks in the near future.

Therefore, the current efforts in academia and the industry are worthy, and all IoT devices would benefit from hardware-based security with strong guarantees.

Do it Pip's way instead As the studied solutions did not satisfy our requirements, we reversed the perspective and looked the other way around. We applied Pip's partitioning scheme to all mentioned systems. As the scheme is quite generic, it revealed disparities in the partition deployment and their inner configuration, for example enforcing globally the $W \hat{} X$ principle, isolating the kernel/OS from the executing tasks, setting up protection stacks, setting up application containers, isolating drivers and modules from the kernel, among other things. More than this, Pip's flexibility gave them more features, like extensible memory and the runtime creation of compartments/sandboxes, which they could leverage to design even more complex applications. Exchanges with the development team of EwoK, and reading TockOS and RIOT-MPU's documentation consolidated our analysis. This successful study gave us the demonstration of the relevance of Pip-MPU.

We did have trouble projecting Pip's security model on some systems as they did not consider a strict memory isolation scheme. Controlled shared memory is an example because shared memory is possible in Pip, but two isolated partitions do not have access

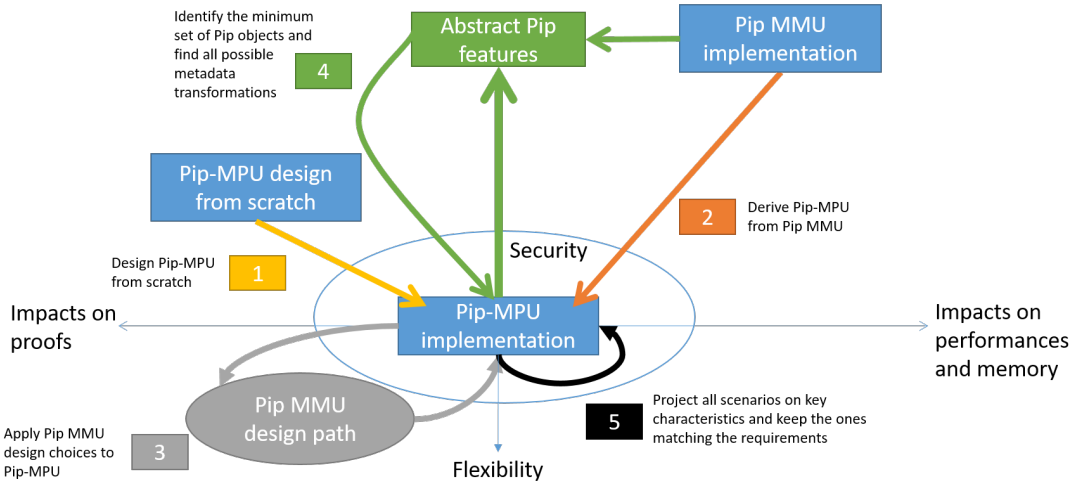


Figure 12.4: Design phases. Several approaches led to the final design, permanently monitoring impacts on the performance, Pip's current proofs, and flexibility while never neglecting the security aspect.

to the shared memory at the same time, they must transfer the memory block around. We created the framework out of these observations, to let systems define their own local policy within the frame of a global security policy, and Pip as a specialisation of this framework. But first we designed Pip-MPU.

Design phases of Pip-MPU

As our initial thoughts (see above) led us to an impasse, we knew we needed first to abstract away from the technical perspective of the MMU. Thanks to the plethora of MPU-based solutions and our thorough study of the MPU, we were also aware of multitude of MPU feature combination that drove our technical choices all along the design phase (see Section 6.4 speaking of the MPU background region and the disabled blocks to name just this one).

Selected approaches to adapt Pip Pip-MPU is born from a mix of several approaches and feedback loops. Our trials considered 1) to design Pip-MPU from scratch 2) impose the new requirements of the constrained environment to Pip (MMU)'s design 3) borrow and apply Pip's design tracks 4) generalise Pip's features and find out which are solely tied to the hardware platform 5) find out the key characteristics of the design and make a selection out of all considered scenarios.

As Pip-MPU's requirements where clear from start, in terms of performance, flexibility, security and proximity with Pip's design, they were closely monitored throughout the whole process.

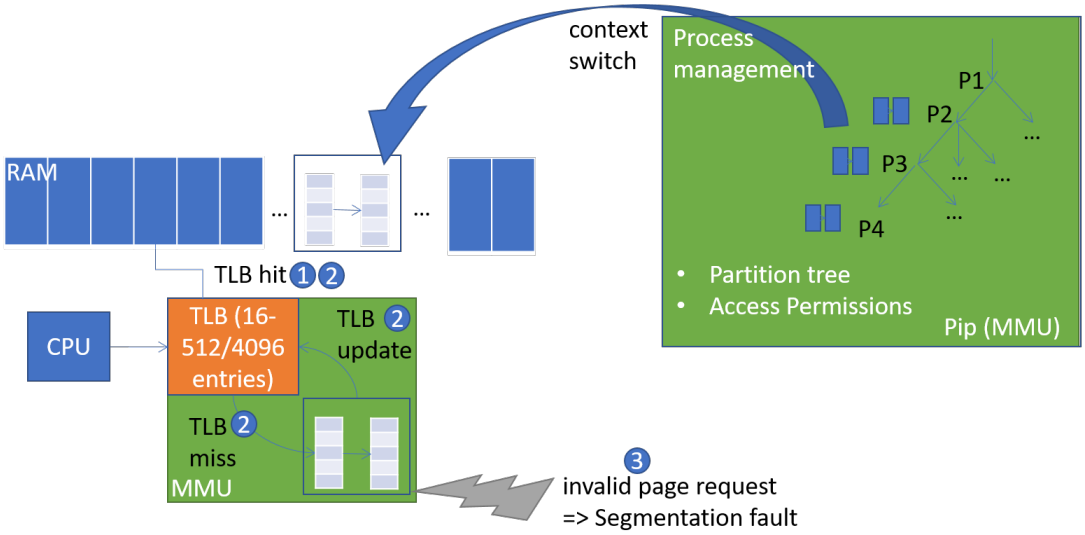


Figure 12.5: Pip (MMU) and MMU reactions after cache hits (1) or cache misses (2-3). In Pip, the TLB handles the cache misses, which is reloaded with pages out of Pip's MMU structure. During a context switch, the MMU is loaded with all the newly active partition's pages.

Scenarios elaboration and corresponding design impacts Out of the previous approaches, we imagined many scenarios from scratch or transforming Pip, while we already knew we would need memory blocks, we knew all the features of the MPU and how existing solutions leveraged them and approached the technical issues, and we knew we had metadata structures to protect.

The direct adaptation from Pip was not obvious and forced us to revise Pip's concepts, like the `cut` ability, the source and target of the services or even the definition of a partition. With the MMU hardware, the system starts with several hundreds to millions of MMU pages, while the MPU does not have this luxury. Because static configurations were out of plan in our design, we imagined the concept of `cut` to break/mine subblocks out of initial memory blocks. Again, several solutions rose. For example, we considered first the root partition as a special partition (it is already special being the only existing partition at boot) that would be the only partition to have the cut ability, while other partitions would need to request cuts from it. Instead, we imagined the `cut` could only be performed by the parent partition on the child partition. Alternatively, we also imagined that any partition would be able to `cut` at will and Pip-MPU would echo back the `cut` in the ancestors only when a metadata structure needed to be protected. And step-by-step we concluded in `cutMemoryBlock` and the concept of an *MPU cache* replacing the TLB of the MMU as illustrated in Figures 12.5, 12.6. Only later, another paper spoke about the same concept [195] (MPU region virtualisation).

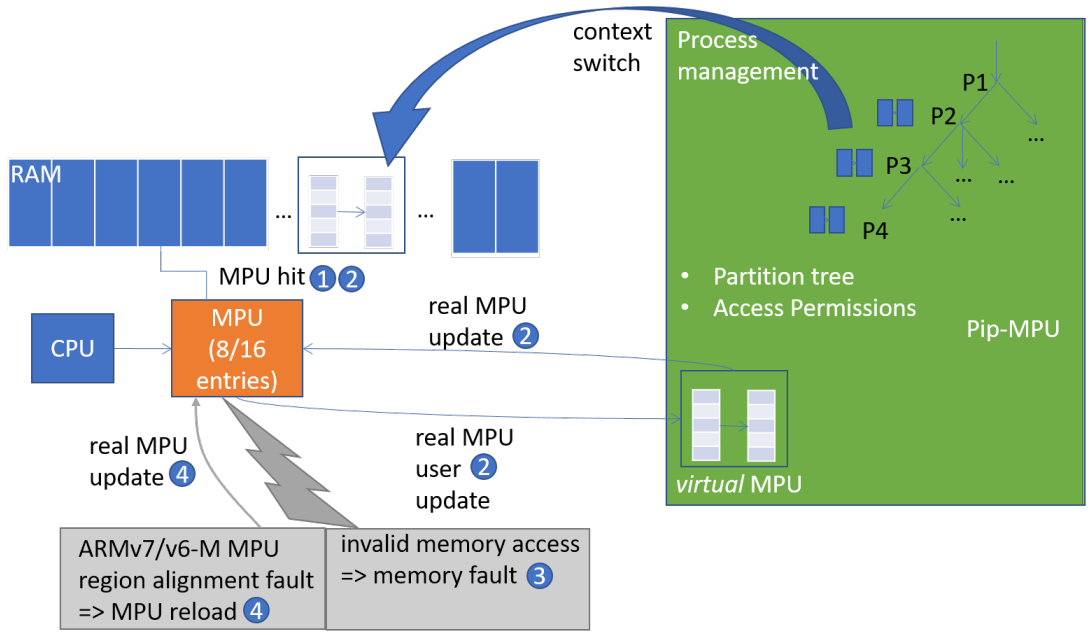


Figure 12.6: Pip-MPU and MPU reactions after MPU hits (MPU correctly configured) (1), MPU updates (2) or memory faults (3-4). In Pip-MPU, the MPU is viewed as cache, like the TLB in Pip (MMU). Indeed, neither the TLB nor the MPU dispose of the entire set of memory pages/memory blocks. However, memory faults are not like TLB misses because the process of reloading the MPU is manual in Pip-MPU while it is hardware driven for the TLB in Pip (MMU). A similar working is the automated MPU reloading proposed in the framework, but was not retained for Pip-MPU. When a memory fault occurs in Pip-MPU, it is because of a illegitimate access (3). Exception is made with the ARMv7-M implementation where the MPU is automatically reloaded (4) when detecting memory faults due to the MPU region alignment and size constraints (see Section 6.3.3). During a context switch, the MPU is loaded with the MPU cache (the set of active blocks).

In addition to that, this led us to reconsider the `prepare/collect` system calls in their current implementations, with the extended feature to call the service on the current partition, notion which is absent in Pip as shown in Figure 7.2. Moreover, we questioned as far as the definition of a partition. In Pip, `createPartition` sets up the first MMU indirection level to all metadata structures. The user must then provide as many pages as structures. Instead, in Pip-MPU, `createPartition` only sets up the PD structure. Equivalent to Pip at that moment, no memory blocks/pages are shared with the new child, so the memory space is not yet completely defined. The `prepare` service completes the partition initialisation phase in both systems.

This path of reflection compelled us to abstract away the technical details of the MMU and generalise Pip's concepts. At the same time, Pip already showed us, I guess involuntarily, the notions that really mattered. Indeed, in Pip, all structures stand alone, and not merged like is done in Pip-MPU. The MMU structure is directly used by the MMU hardware as pointer to configuration tables while the MPU needs to be configured register by register. Separate the MMU structures from the rest showed a distinction between the memory space definition (MMU structure), the relations between the partitions (Shadow 1, used to reconstruct the partition tree), optimisation structures (Shadow2 and LinkedList) and the superstructure that makes the bound between all the structures. Thus we identified the latter structure, MMU structure, and Shadow1 structure as essential for Pip. We crafted new scenarios of all possible combinations with these structures and the Shadow Cut. We extracted the characteristics of the scenarios and map them on a single figure to understand the relations between them and to reveal the missing combinations, as shown in Figure 12.7.

Each scenario has also been assessed in terms of impacts on the requirements. For that, we challenged each scenario with our requirements in an informal way. As can be seen in Figure 12.8, candidate scenarios emerged.

Adaptation path and selected characteristics As said earlier, sticking close to the current metadata structures has always been the target as there was hope to reuse the formal proofs of the services. To decide which candidate scenario would suit the best this condition, we unfolded the relations between the scenarios starting from Pip (MMU)'s design. The Figure 12.9 gives us the adaptation path from Pip (MMU) to one of the candidate scenarios, so the number of transformations needed to reach each one of them.

We chose a scenario close to the current design and that exposed the greatest flexibility. Furthermore, proofs are easier with simple structures as shown by past projects in the team. Formal verification is covered in the next part.

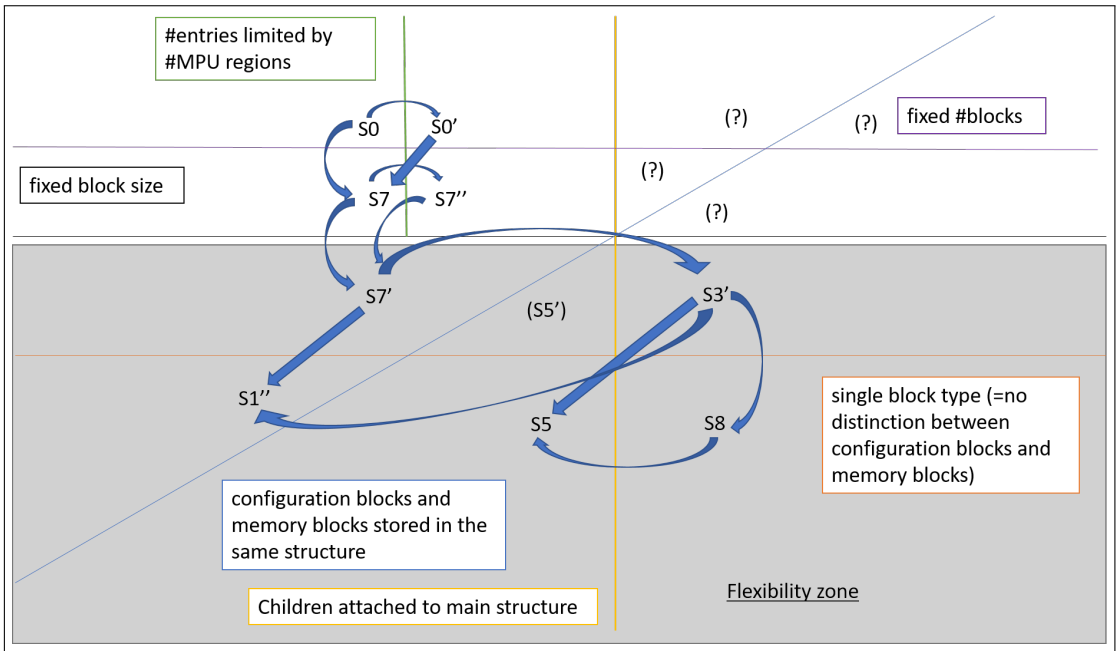


Figure 12.7: Scenarios mapped to their characteristics at some moment in time. Missing combinations are represented by (?). The arrows indicate a one-step transformation of the initial scenario to reach the final one. The flexibility zone is the set of characteristics enough to ensure flexibility in the design.

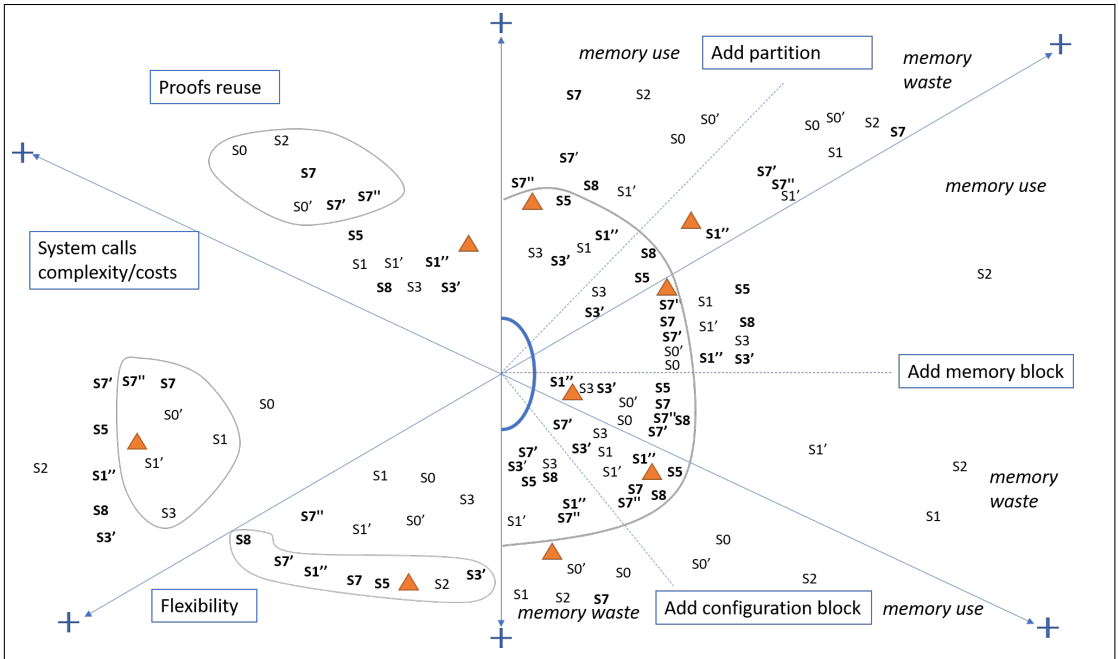


Figure 12.8: Impacts of each scenario on memory footprint, flexibility, system call complexity and proofs reuse. Candidate scenarios are in bold. The circled areas show the desired range. Smaller values are closer to the centre while higher values are to be found in the peripheries. The orange triangle represents the final retained scenario.

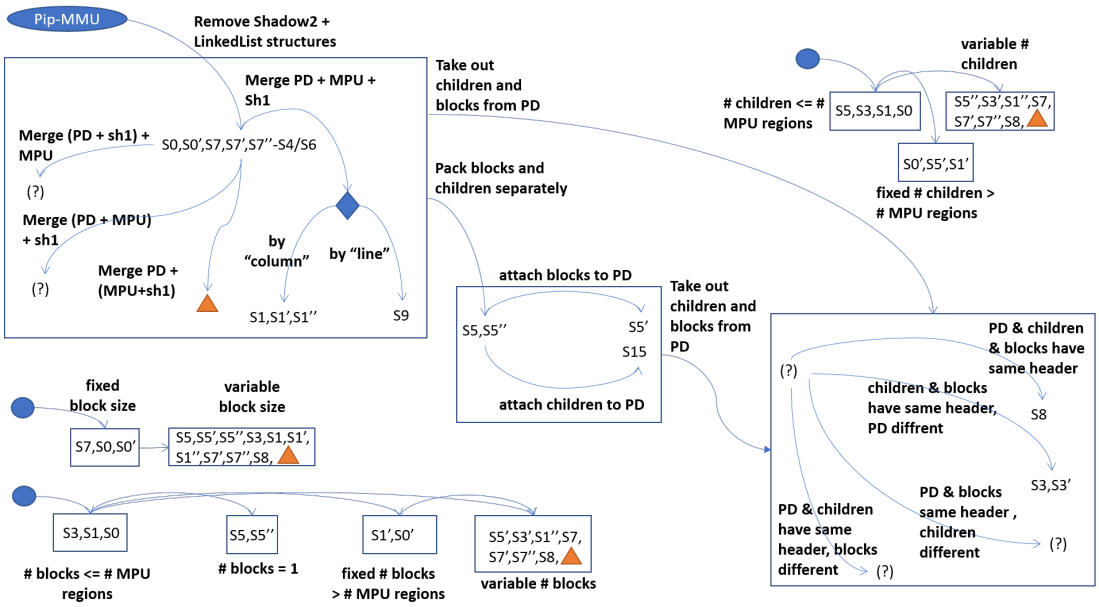


Figure 12.9: Adaptation path from Pip (MMU) to the selected scenario describing the design of Pip-MPU. The figure should be read as follows. The oval shape "Pip MMU" is the starting point of all transformations. Each transformation changes a parameter in the metadata structures (how the structures are merged and which information they hold). The blue circles all around describe flexibility characteristics: fixed or variable memory block size, number of memory blocks in a memory space, number of children in a memory space. Hence, each transformation can be linked to a characteristic, meaning several adaptation paths could lead to the same transformation, however the scenarios will differ in characteristics, which will have an impact on the requirements as discussed in Figure 12.8. The orange triangle represents the final retained scenario and gives us the transformation to make on Pip (MMU) to reach it.

Summary

This long section puts choices and discoveries back in chronological order.

First, the original intuition is that IoT devices are not protected as they could and should be. However in the recent years, massive and powerful attacks were performed remotely on low-end IoT devices. In the end, servers and computers are usually the primary aim of the attack, and need to be protected. Strengthening low-end devices would benefit them as well and simplify their attacker model because IoT devices might have privilege access to protected networks and are vectors of indirect attacks, with lateral compromise of servers and high-end devices. With the evolution of the market landscape, because legacy embedded systems join the IoT, we observe a shift in development mentality, but security is still not their first priority. This dissertation demonstrates that the industry has the available technologies to secure the ecosystem.

Our adaptation path shows there was a necessity to extract the fundamental characteristics of Pip, to understand the hardware, to analyse how other systems solved some common issues to fit their use cases. Our journey showed Pip needs access control mechanisms and metadata structures to register the partition tree. Pip-MPU could have been designed only in software, instead of relying on the MPU hardware, but this would have involved heavy impacts on performances and less reliable implementations. While we did imagine Pip-MPU from scratch, we always had the requirements to stay close to Pip's current design and behaviour that implied previous choices (for example that shared memory is still accessible to a parent partition when the memory block is given to a child partition). Things would have probably been different if Pip (MMU) did not exist yet, and that Pip-MPU were to be the first instance of Pip, which could have been more difficult to adapt Pip (MMU) starting from Pip-MPU. Or, in a different way, maybe Pip (MMU)'s design were to change during our design phase and would add new requirements on our design or, in the opposite, release some requirements by mutating Pip (MMU) to something that would ease the adaptation. We could also have chosen to base our design on an existing MPU-based solution and adapt this one to fit Pip (MMU)'s design, however, their lack of flexibility would not have given us the pursued genericity.

Furthermore, the switch from MMU to MPU was not clear from start as no other example existed to show us the way. Some projects already thought about it though, but the other way around so from MPU to MMU like Zephyr that considers the MMU as an MPU which entries are unlimited, still ongoing work [191, 190]. The position statement in Pip-MPU's current design is to state a page is a memory block of a fixed size and their number is limited for each partition. We could see it differently: a memory block is made of several pages of fixed size, or that a memory block and a page is just a set of contiguous memory addresses (approach taken during the formal verification, disclosed

in the next part of this dissertation).

Finally, the figures shown in this chapter express a lack of analysis tools in this area. Our customised approach might have missed important combinations, or in the contrary assures us that we were converging to an acceptable solution. The team's know-how regulated our choices, I tried here to present some of them that were the most important for the reader that would also need to switch to a more limited hardware platform.

Insights and learnings on the formal verification of Pip-MPU

Pip-MPU's trustworthiness is assessed by formal proofs. The formal verification of Pip-MPU follows the refactoring of Pip's design. And because the design had been deeply changed, so had the proofs. Like previously done, this section gives more insights on the formal verification process of Pip-MPU. Future formal verification projects, linked to Pip or not, can find here the applied methodology, with hopes and retrospective workarounds, and what I would have done to accelerate, improve, simplify the process. It gives me the possibility to light up, in an informal manner, some left-aside aspects of the topic.

History and methodology

Pip-MPU's design was created, with the aim to recycle much of Pip's design, followed by the implementation, tests, and benchmarks of Pip-MPU. Only after that came the verification process. Officially. In fact, the verification process started even before Pip-MPU's design.

Because of calendar constraints, the first weeks of the PhD started with an express training on the formal verification process followed in Pip. Design choices, proof techniques, proofs organisation, the discovery of the Coq proof assistant, everything I needed to know on the current project, carefully laid on paper until the time I would need these learnings again.

Furthermore, the implementation of the services was directly developed within Coq, following Pip's workflow, which demonstrates the formal verification was the final goal. We have been able to apply Pip's workflow and we strove to change the models as little as possible to still recycle proofs with light changes (I intentionally let old terminologies, like *page*, in the proofs to pinpoint exact places were proofs were fully recycled). The followed methodology was first to implement the services in Coq from the Python prototype, create the underlying primitives, adapt the hardware model and adapt or remove all primitives that manipulated the old model (remove all virtual memory from the code base). Then I restructured the files to split different intertwined verification

lemmas (while still trying to be as close as possible to Pip's code base), adapted the security properties and started the horizontal exploration.

On their side, proof metrics arrived early in the proof process, with the need to communicate my progress despite the unknown and lack of expertise and experience on the subject of both my managers and myself. The global proof status naturally gave some sort of idea of the progress during the fast pace horizontal exploration. However, the vertical exploration took more time, but weekly reports did not change in frequency. The proof dashboard got enriched with finer proof analysis and I clearly saw proof patterns matching the expression of the properties, which finally gave birth to the proposed proof complexities.

Formal verification from scratch...

When the time arrived to apply the knowledge learned in the first weeks, the world had the time to stop and restart several times because of the pandemic, and notes and tips referred now to obscure notions I didn't understand the meaning anymore. So everything had to be built up again. In addition to that, we had some primary hopes to recycle a lot of proof in the beginning (as discussed in Section III), but hopes became vain from the moment we modified the hardware model, as it burst the whole formal baseline.

The formal verification process demonstrated again the expertise and know-how associated with it. With Pip-MPU, I rediscovered a lot of lost knowledge due to team members pursuing other careers. The learning curve is steep to become a proof expert, proofs are hard, environments different from software development, poor automation in the case of theorem provers and intuitions hard to implement without digging into the technical details (and a lot of time manuals written for mathematicians) to set them up. Other formally-verified projects also report the high cost of formal verification [108]. Nevertheless, this thesis demonstrated it was possible for a formal verification noob with computer engineering background to gain enough skills to conduct long and challenging proofs (for me). Tony Hoare already warned programmers about it: 'will it be too onerous for a programmer to supply the necessary assertions and lemmas needed for their proofs ? Yes indeed' [91]. Moreover, Pip benefited from a toy system to train the proof skills and discover the first consistency properties [99]. Pip-MPU did not take this course and faced rapidly many challenges to construct the proofs.

I understand all my notes again now, that can only be deeply understood by those who walked on Pip's formal verification path. I see now how symmetrical the verification path of Pip and Pip-MPU have been, but could not have been projected earlier, as many alternative paths exist. I guess we faced the same constraints at some similar moments in the verification process that made us take the same direction. Some alternatives have

been explored though, sometimes with inconclusive results.

...led to failed attempts...

My initial plan was to enrich the page definition so to become variable in size. This revealed to be not enough, since Pip's code base was completely built around the MMU virtualisation, which does not exist in Pip-MPU. However, I was still hoping to form a similar structure for the security properties. Because, all in all, the page just missed a field to register the end address and they would be the same. In fact, it was not so simple.

At issue were distinct page and block identifiers. In Pip, a page has a unique identifier in the whole system and pages holding Partition Descriptors can be directly identified. In Pip-MPU, a block is only identified locally to the metadata structures and this local identifier must translated in a global perspective to find the associated Partition Descriptor if any. This is because the system calls just act locally so there is no need for a global identifier and mainly to accelerate the search for the block in the system calls.

Back to the security properties, the local identifier could not capture the globalism of pages. What if a block was cut in a child, how to check if a subblock was in the parent with different identifiers? I was first tempted to create another abstraction of the block, a sort of plain memory region, rejecting any unneeded block characteristics like access permissions. But I was still missing how to express block overlapping. As a result, I downgraded the security properties to the granularity of the address. The security properties are still expressed the same, looks the same, means the same, but at a finer granularity.

When the time came to prove the security properties, I discovered the issue with lists. Indeed, I previously crossed the free slots list in `addMemoryBlock` service, however this was about to remove an element from the list. Then, all the elements still in the list satisfied all properties since they were there before, and I could propagate from the beginning the sublist as I knew it would be the final one. In the security properties, it was a different situation since this time lists were extended with new elements and these lists only appeared in these properties, so I did not encounter them before in the consistency properties. I could not apply my proof pattern anymore because I had to deal with mutable sets, whereas it was enough to just consider single elements in the memory state that could be deduced from any modified state, until now. Even at the proof checkpoints, proofs were (impossible) difficult to conduct because impossible to link with a known state.

I then shifted to an instruction-by-instruction list propagation strategy. From a

single entry modification, I could project the modified list *in relation with* the previous list. I understood at that point why all properties were proven instruction-by-instruction in Pip. Specific entry modification lemmas for the properties had been developed for this. However, it supposes the possibility to prove them at each instruction, which, in the case of Pip-MPU was not possible and led to the introduction of the checkpoint strategy. Instruction swapping could have been possible so to effectively satisfy them, with the Pip's co-design approach and as promoted by Edsger Dijkstra: 'the programmer should let correctness proof and program grow hand in hand' [59]. I still consider the checkpoint strategy followed in Pip-MPU more efficient, because it saves time in the first pass and allows to create reusable elements saving proof effort, where in Pip we would probably had duplicates proof script in that code portion.

The introduced list propagation lemmas were numerous, sometimes intertwined like *getAccessibleMappedBlocks*, and *getMappedBlocks*, both relying on the *getKSEntries* list applying different flag filters. It also implied a lot of subsidiary consistency properties, and the whole set now resembled much more like Pip's consistency properties, which consolidated the proof strategy shift. It also increased a lot the proof effort because a lot of modification instructions involves the lists, as witnessed by the number of '*' in the impact score Table 12.6.

...and proof development tips and tricks.

Despite similarities, obviously numerous for an adaptation, Pip-MPU followed another proof course and I share here some know-how tips, some of them have already been introduced earlier but are here clearly mentioned.

Gather global properties at checkpoint The checkpoints are like reset points in the proofs. The consistency properties are proved for that particular state, intermediate or final, like the initial state. It is a privileged instant to gather global knowledge about the context that is extensively used thereafter, and thereby avoiding duplicates in the proof script.

Ease the maintenance The context is made of hypotheses that are automatically named by Coq if not explicitly by the proof developer. The several iterations imply to go back and forth on the proof script which modify the context at each passage. Hence, references to hypotheses change frequently. From the second passage, I found it convenient to name properties that are used frequently which facilitates the maintenance. Sometimes it is difficult to name them, for example because of a very long hypothesis or properties that might change formulation. In these cases, the automated references are kept but I add in parenthesis hints to find the hypothesis again if necessary for a next iteration.

Low-level invariants HAL invariants are simple to prove. The proof script can be extensively reused with only light adaptations. One possibility, not followed in the course of this work, would be to prove all HAL primitives at once because they are independent of the context. However, this would slow down the proof pace for a particular service.

Modularity The more generic the code is, the more abstract are the proofs and the more they are reusable in different contexts. This is counter-intuitive for a programmer that just wants an executable code. However, it can save a lot of time for the proof developer by reusing the proven brick and it avoids code and proof duplicates which facilitates maintenance. Furthermore, it clarifies the operation intentions and avoids mistakes, for example forgetting to set a field that has no security implications like a block's access permission rights.

Enriched memory types Pip-MPU presents more memory types than Pip (MMU). They are used to easily discriminate the cases without having to search in the context for implications resolving the type. Hence, consistency properties which shape the relationships between kernel elements and combined with enriched memory types simplify the proofs.

Meta-properties Pip (MMU) already used meta properties that gather several properties that have a meaning in a certain context. For example, the meta-property $isFreeSlot$ carries the information that, if a *Blocks* entry and associated $Shadow1$ and $ShadowCut$ entries are valid, then all fields hold default values. They are naturally constructed in the proof flow and usually detected when writing down each instruction and identifying which property should be propagated and retrieved for a later instruction.

On-the-fly property adjustments At rare moments, I discovered that some properties are actually false or expressed in a way that they cannot be easily invoked. In such a case, I copy-pasted the property statement at the beginning of its proof, modified it in a better way, and asserted it. The new proof obligation was to prove the corrected property inheriting from the real proof context of the previous property. This possibly implied to temporary admit some other properties. Once satisfied with the modified property, and its proof is passed, I retroactively modify the property from the set of consistency properties, which in turn might imply correcting the proof at other points.

Numerous small properties In search for the consistency properties, they changed shape a lot. Finally, I found it easier to split the properties in atomic meaning, rather than an umbrella property with many of them. First of all, this facilitates the checkpoint

strategy by actually having the possibility to split them. But more importantly, bigger properties lose in precision, in the sense they can implicitly imply combinations that were not expected, and situations might show they are false. Furthermore, these bigger properties can be deduced by proving the smaller ones that were detached.

Code and proof co-design I said earlier Pip-MPU did not follow the co-design approach of Pip. This statement is partially false (Dijkstra has been heard). Indeed, I voluntarily added a check in the code to spare a logical connexion and so a consistency property. The accessible flag is in fact linked to the present flag: a block cannot be accessible if it is not present. However, a single flag checking did not impact so much the performances but spared many proofs, because it is a read instruction with negligible proof effort, even if the proof complexity would have been low (one additional check for no proof effort).

Furthermore, we abandoned some design ideas to facilitate the proof. Typically, we considered first a doubly linked free slots list, but this would have added an equivalence property on lists that we would gladly avoid.

Isolated proof scripts for the security properties and each proof element The security properties are the ultimate goal of the verification. The other proofs are just there to form the proof context out of which the security properties can be proven. It then makes sense to isolate them from the rest.

In addition to that, it was actually necessary. We already knew from Pip (MMU) that Coq had trouble dealing with proofs that consumed more than 32 GB of RAM. Well, my computer had only 16 GB, but even with 16 GB of swap space, Coq crashed multiple times and before that became very slow to check a single tactic. It is then necessary to filter out as much as possible from the proof context, simplify it, and extract proof elements in different files.

Proof testing and final proof convergence technique I see the proof process as a test-driven development towards the security properties. There is also a way to test the services satisfy the properties by directly jumping to the security property proof script. It consists in reporting the state modifications, expected list propagation and the checking phase proof context, and trying to prove the properties. The proof is complexity-driven, meaning I explore the hardest path to prove the properties by assuming many properties true (with the `admit` tactic), letting aside proof portions with low complexities. Then, I prove the hardest admitted lemmas, still leaving very low complexity proof properties, until reaching a point where I have confidence in all the left-aside properties. I go back in the other proof scripts and do the same. This means I went through the whole proof

once, know the upcoming difficulties that I try to minimise everywhere, explored all possible cases, and slowly converge towards a full verification. Hence, the confidence shows an exponential convergence.

Git Git has been extensively used during the proof development and especially in the first stage to span over all check phases of all services. Each service had their own Git branch and one branch had common lemmas to all of them. The work could be distributed across branches and synchronised with the one containing the common lemmas. This methodology had the advantage of isolating each service so that proofs would not break everywhere due to changes when reaching the vertical exploration phase. However, it also had the unpractical disadvantage of letting partial proofs in distributed branches, not synchronised with the latest advances and thus increasing the complexity to merge them all once a service has been fully verified. It would have been probably better to merge all partial proofs together after the horizontal exploration phase, because they were synchronised with the common lemmas branch. And then, the branch of the first proven service becomes the master proof branch, whence the common lemmas branch can synchronise. Afterwards, we could continue the vertical exploration with another service in a separate branch, that joins the master proof branch at proof completion, and synchronises with the common lemmas branch.

...which turned out to become a sort of game...

Formal verification could be very recreational in its way, and there is a sort of addiction and reward to seeing the proof progress tactic by tactic. Sometimes though, the "boss monster" is too strong and one lacks a magic potion to facilitate some moments in the proof. Coq's heuristics are too limited for the junior proof developer I am. My impression is that it could be a tedious, intellectually highly-demanding process, even though I had experts in the field around me.

...useful in real-life.

Formal verification has a lot of benefits for Pip-MPU. To start, it spotted a critical bug (*cf.* section 10.4.5) despite a 99% code coverage by unit tests in Python, later translated in C and part of the final code base, with several security tests, and a running prototype for months. Indeed, functional correctness was almost completely tested, however I only partially tested the isolation property because formal verification was expected. This bug would certainly have been found earlier with extended security tests because it was directly related to the security properties.

However, if I affirm this, I am not sure about other more sneaky bugs that may still

be in the code because Pip-MPU has not been fully formally verified yet. Indeed, even Pip (MMU) reported bugs found during the verification process, which gives sense to the process, but raises concerns about large projects where formal verification could not even be applied because of the huge complexity. As is common knowledge in our research team given past projects, but also known in other external projects, there is no real certainty until formally proven (or more extreme, is worth nothing).

Formal verification also forces us to make explicit design choices in the proofs, including implicit, obvious, or forgotten ones. It pinpoints the minimum elements on which the proof holds, which gives another view on the code development process.

Furthermore, this calls design certitudes into question. For example, we choose some structures in the design phase that must be assessed again during the formal verification process regarding proof impacts.

It also allows to analyse the final proof context and what hypotheses are effectively used to conduct the proofs. Hence, in an orthogonal direction, it highlights the most impactful instructions, which should match our proof complexity analysis.

Personal thoughts

This last section steps back even further on the PhD and related topics lived from the inside and looked from the outside.

According to my experience with formal verification in this work, I believe formal verification tools still need improvements for massive and natural adoption in any software projects. There seems to be more maturity in proving runtime errors than higher-level properties. In my opinion, tools like Coq should include more automation, a less steep learning curve which is also linked to a simplification of projects to verify, and maybe other complementary tools or interfaces like graphical ones. This dissertation also narrates my learning curve and how I approached the research questions with little knowledge of formal verification. Hence, it shows the necessary resources to successfully undertake this work exist, but that would not have been possible without punctual support from my team and their expertise on the tools and past projects.

In another perspective, I lacked of reasoning methods and tools for the design and the formal verification, so I developed my own methodology and mental framework presented previously. This was also exacerbated by current physical equipment that is too limited in my opinion, like blank boards, papers, numerical tools. As proof development is much about human intuition, not needed to be explicitly exposed for code development, I feel novelties in this sector would facilitate the manipulation of thoughts and bring to light unconscious mental processes.

Another thing is that this work is under an open-source licence. Much of it would

not have been done without open-source tools and developers: code inspiration for tricky parts, review of existing MPU-based systems, benchmarking tools, Integrated Development Environments (IDEs), programming languages... Especially for system reviews, nothing is better than digging into the source code because the documentation might be weak in some parts, not up-to-date or not existing at all. It shows the power of open science and long-term gifts from the past, which continues with this work.

However, I think a flaw would be to rely too much on standards, which increases the number of eyes and community efforts that concentrate means to control, develop and maintain a code base for the many. Counter-intuitively, it weakens the shared component because a vulnerability discovered directly impacts an enormous number of devices under a great number of malicious eyes. Formally expressed by Milosch Meriac: $Value\_of\_bugs = number\_of\_installations * device\_value$ [123]. We rely more and more on connected objects, which can be found in critical devices such as pacemakers, and as such gain value for widespread deployments. A large part of low-end devices are also low-cost devices under the time-to-market pressure and limited budgets that are the playground for security researchers but also cyber attackers.

About knowledge sharing, the COVID-19 pandemic opened access to a lot of scientific resources like articles, exchange platforms and open-access conferences. Even if scientists were alone at home, it seemed to me there was much more material to nourish everyone's studies. However, it also meant less informal interactions with researchers and in my case a sole physical conference over the three-year PhD period, which besides occurred at the end because the pandemic just started at the same time as the PhD. With the pandemic pullback, that knowledge sharing period is over. While I do think there are still great researchers nowadays, it comes to me that researchers have an unequal access to knowledge, unequal fundings, publication number and time pressure, weird rankings, heterogeneous reviewing process and committees, that are not aligned with universal access to knowledge, fair science, massive research evolution and society retribution with impactful discoveries.

The works undergone during this PhD have very deep roots, from OS design in the 60s, to formal verification tools in the 80s, improved by fully formally verified kernels in the 2000s and processors from the 2010s, getting inspired by a protokernel from a few years ago, but also Ancient Greek logical foundations and certainly way back. There is a gift in science from past generations to the current ones, and from the current ones to the future generations. Much of this work inhabited me for three years, equally strengthened and improved by my supervisors; and time has come to pass it on. While I guess this is a drop in the ocean of science, I hope it will resonate with other rain drops and participate in a greater long-term wave.

Finally, this has been the joyful experience of a PhD student. I stress it here as a final note because the choice of pursuing the studies with a PhD was not obvious. I have seen and met more stressed students than serene students, that were sometimes depressed, engaging their soul in an existential crisis period with doubts and darknesses. While this PhD was not easy to conduct, particularly because of the pandemic and the uncertainty it created, the logistic considerations to be in two research teams separated by a distance of 300km with constraints on both sides with associated moving ins and outs or the lack of knowledge in formal verification, I do not regret this choice. It became obvious.

Closing

The curious schoolboy of the 'Foreword' finally arrived home after a long but pleasant day at school and evening activities, the head full of learnings and new ideas. That night, he will succeed to break his computer's password restricting his computer time, and enjoy the evening even more. We know the rest of the story. ■

Bibliography

[1] 2XS Team. *Website of: 2XS team (CRIStAL laboratory)*. https://www.cristal.univ-lille.fr/2XS/. [Online; accessed October 10, 2022]. 2022.

[2] Naif Saleh Almakhdhub et al. "BenchIoT: A Security Benchmark for the Internet of Things". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019), pp. 234–246. DOI: 10.1109/dsn.2019.00035.

[3] Mahmoud Ammar et al. "SV—The Security MicroVisor: A Formally-Verified Software-Based Security Architecture for the Internet of Things". In: *IEEE Transactions on Dependable and Secure Computing* 16.5 (2019), pp. 885–901. ISSN: 1545-5971. DOI: 10.1109/tdsc.2019.2928541.

[4] Andrew Tanenbaum. *Keynote on: Smaller is Safer (A. Tanenbaum)*. https://summit.riot-os.org/2020/blog/speakers/andrew-tanenbaum/. [Online; accessed October 10, 2022]. 2022.

[5] June Andronick et al. "Large-Scale Formal Verification in Practice : A Process Perspective". In: (2015).

[6] ANSSI. "La défense en profondeur appliquée aux systèmes d ' information". In: (2004), pp. 1–51.

[7] ANSSI. *MultiApp V4 JavaCard Virtual Machine*. https://www.ssi.gouv.fr/certification_cc/multiapp-v4-javacard-virtual-machine/. [Online; accessed October 10, 2022]. 2022.

[8] Apple. *Website of: MacOS Monterey*. https://www.apple.com/macos/monterey/. [Online; accessed October 10, 2022]. 2022.

[9] ARM. *Arm Architecture Reference Manual Supplement - Morello for A-profile Architecture Section 1.2: CHERI protection model*. https://developer.arm.com/documentation/ddi0606/latest. [Online; accessed October 10, 2022]. 2022.

[10] ARM. *Capabilities for memory security*. https://developer.arm.com/documentation/den0132/0100/Overview/Capabilities-for-memory-security. [Online; accessed October 10, 2022]. 2022.

[11] ARM. *mbed uVisor for mbed Overview High Level Design Overview Memory Allocation System mbed uVisor for mbed*. 2016.

[12] ARM. *Website of: ARM Cortex-M for Beginners (Whitepaper)*. https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-52-96/White-Paper-_2D00_-Cortex_2D00_M-for-Beginners-_2D00_-2016-_2800_final-v3_2900_.pdf. [Online; accessed October 10, 2022]. 2022.

[13] ARM. *Website of: Arm Cortex-M Processor Comparison Table*. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Cortex-A%20R%20M%20datasheets/Arm%20Cortex-M%20Comparison%20Table_v3.pdf. [Online; accessed October 10, 2022]. 2022.

[14] ARM. *Website of: ARMv7-M Architecture Reference Manual*. https://developer.arm.com/documentation/ddi0403/ee/. [Online; accessed October 10, 2022]. 2022.

[15] ARM. *Website of: ARMv8-M Memory Protection Unit Version 2.0*. https://static.docs.arm.com/100699/0200/armv8m_memory_protection_unit_100699_0200_en.pdf/. [Online; accessed March 09, 2021]. 2017.

[16] Arm. *Website of: Arm Cortex-M differences (Arm)*. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0/. [Online; accessed March 27, 2020]. 2020.

[17] Arm. *Website of: Morello Board Program*. https://www.arm.com/architecture/cpu/morello. [Online; accessed October 10, 2022]. 2022.

[18] Arm. *Website of: Morello Compartment Demo*. https://git.morello-project.org/morello/android/vendor/arm/morello-examples/-/tree/morello/mainline/compartment-demo. [Online; accessed October 10, 2022]. 2022.

[19] Arm Limited. *Introduction to the Armv8-M architecture*. Version 2.0. 2017.

[20] Arm mbed. *Website of: Mbed OS*. https://os.mbed.com/mbed-os/. [Online; accessed November 20, 2020]. 2020.

[21] David Aspinall and Cezary Kaliszyk. "Towards formal proof metrics". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9633 (2016), pp. 325–341. ISSN: 16113349. DOI: 10.1007/978-3-662-49665-7_19.

[22] Emmanuel Baccelli et al. "RIOT OS : Towards an OS for the Internet of Things". In: (2013), pp. 2453–2454.

[23] Emmanuel Baccelli et al. "Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things". In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2018* July (2018), pp. 504–507. DOI: 10.1109/PERCOMW.2018.8480277.

[24] R. William Beckwith, W. Mark Vanfleet, and Lee MacLaren. "High Assurance Security/Safety for Deeply Embedded, Real-time Systems". In: *Proceedings of the Embedded Systems Conference.* (2004).

[25] Ryad Benadjila et al. "Wookey: Designing a trusted and efficient USB device". In: *ACM International Conference Proceeding Series* (2019), pp. 673–686. DOI: 10.1145/3359789.3359802.

[26] Quentin Bergougnoux. “Co-design et implémentation d’un noyau minimal orienté par sa preuve, et évolution vers les architectures multi-coeur”. PhD thesis. Université de Lille, 2019.

[27] Bleeping Computer. *Website of: Hacker used ransomware to lock victims in their IoT chastity belt*. https://www.bleepingcomputer.com/news/security/hacker-used-ransomware-to-lock-victims-in-their-iot-chastity-belt. [Online; accessed OCtober 10, 2022]. 2021.

[28] D Bolignano. “Formally Proven and Certified Off-The-Shelf Software Components: the Critical Links for Securing the Internet of Things”. In: *CESAR* (2016). URL: https://www.cesar-conference.org/wp-content/uploads/2017/06/Actes_Cesar_2016.pdfpage=109.

[29] Pauline Bolignano. “Formal Models and Verification of Memory Management in a Hypervisor”. PhD thesis. Rennes 1, 2017.

[30] Guillaume Bouffard and Léo Gaspard. “Hardening a Java Card Virtual Machine Implementation with the MPU”. In: (2018).

[31] Campus Cyber. *Website of: Campus Cyber*. https://campuscyber.fr. [Online; accessed October 10, 2022]. 2022.

[32] Abraham A. Clements et al. “ACES: Automatic compartments for embedded systems”. In: *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 65–82.

[33] Abraham A. Clements et al. “Protecting Bare-Metal Embedded Systems with Privilege Overlays”. In: *Proceedings - IEEE Symposium on Security and Privacy* (2017), pp. 289–303. ISSN: 10816011. DOI: 10.1109/SP.2017.37.

[34] Abraham A. Clements et al. “Protecting Bare-Metal Embedded Systems with Privilege Overlays”. In: *Proceedings - IEEE Symposium on Security and Privacy* (2017), pp. 289–303. ISSN: 10816011. DOI: 10.1109/SP.2017.37.

[35] Common Criteria. *Common Criteria Portal*. https://www.commoncriteriaportal.org/. [Online; accessed October 10, 2022]. 2022.

[36] COMPAS. *Website of: COMPAS 2020 conference*. https://2020.compas-conference.fr/. [Online; accessed October 21, 2022]. 2022.

[37] COMPAS. *Website of: COMPAS 2021 conference*. https://2021.compas-conference.fr/. [Online; accessed October 21, 2022]. 2022.

[38] COMPAS. *Website of: COMPAS 2022 conference*. https://2022.compas-conference.fr/. [Online; accessed October 21, 2022]. 2022.

[39] Contiki-NG. *Website of: Contiki (Github)*. https://github.com/contiki-ng/contiki-ng. [Online; accessed October 28, 2022]. 2020.

[40] Le Coq et al. “Le Coq’ Art (V8)”. In: (2015).

[41] David Costanzo, Zhong Shao, and Ronghui Gu. “End-to-end verification of information-flow security for C and assembly programs”. In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 648–664. ISSN: 0362-1340. DOI: 10.1145/2980983.2908100.

[42] CRIStAL. *Website of: CRIStAL laboratory*. https://www.cristal.univ-lille.fr/en/. [Online; accessed October 10, 2022]. 2022.

[43] CVE. *Website of : CVE, Linux kernel vulnerabilties*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+kernel. [Online; accessed January 17, 2020]. 2020.

[44] CVE. *Website of: Common Vulnerabilities and Exposures (CVE) database*. https://cve.mitre.org/index.html. [Online; accessed November 18, 2020]. 2020.

[45] CVE. *Website of: Common Vulnerabilities and Exposures database, keyword arbitrary code*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=arbitrary+code. [Online; accessed November 18, 2020]. 2020.

[46] CVE. *Website of: Common Vulnerabilities and Exposures database, keyword elevate*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=elevate. [Online; accessed November 18, 2020]. 2020.

[47] CVE. *Website of: Common Vulnerabilities and Exposures database, keyword kernel privileges*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=kernel+privileges. [Online; accessed November 18, 2020]. 2020.

[48] CVE. *Website of: Common Vulnerabilities and Exposures database, keyword memory corruption*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=memory+corruption. [Online; accessed November 18, 2020]. 2020.

[49] CVE. *Website of: Common Vulnerabilities and Exposures database, keyword overflows*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=overflows. [Online; accessed November 18, 2020]. 2020.

[50] CVE. *Website of: Common Vulnerabilities and Exposures database, keyword privilege escalation*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=privilege+escalation. [Online; accessed November 18, 2020]. 2020.

[51] CVE. *Website of: Common Vulnerabilities and Exposures database, keyword root privileges*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=root+privileges. [Online; accessed November 18, 2020]. 2020.

[52] CVE. *Website of: CVE, Write-What-Where condition*. https://cwe.mitre.org/data/definitions/123.html. [Online; accessed October 10, 2022]. 2022.

[53] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. "A path to scale up proven hardware-based security in constrained objects". In: *Conférence francophone d'informatique en Parallélisme, Architecture et Système (COMPAS 2020)*. Lyon, France, June 2020. URL: https://hal.archives-ouvertes.fr/hal-03318088.

[54] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. "Compartimentation dynamique imbriquée pour objets contraints". In: *Conférence francophone d'informatique en Parallélisme, Architecture et Système (COMPAS 2021)*. Lyon (en virtuel), France, July 2021. URL: https://hal.archives-ouvertes.fr/hal-03318078.

[55] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. "Evaluation d'une solution d'isolation pour objets contraints". In: *Conférence francophone d'informatique en Parallélisme, Architecture et Système (COMPAS 2022)*. Amiens, France, July 2022. URL: https://hal.archives-ouvertes.fr/hal-03710419.

[56] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. "From MMU to MPU: adaptation of the Pip kernel to constrained devices". In: *3rd International Conference on Internet of Things & Embedded Systems (IoTE 2022)*. Sydney, Australia, Dec. 2022. URL: https://hal.archives-ouvertes.fr/hal-03705114.

[57] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. "Nested compartmentalisation for constrained devices". In: *2021 8th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2021, pp. 334–341. DOI: 10.1109/FiCloud49777.2021.00055.

[58] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. "Perspectives on security kernels for IoT". In: *RESSI (Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information)*. Online, France, Dec. 2020. URL: https://hal.archives-ouvertes.fr/hal-03102252.

[59] Edsger Dijkstra. "The humble programmer - 1972 - Dijkstra". In: *Acm* 15.10 (1972).

[60] Edsger W. Dijkstra. "On the Role of Scientific Thought". In: 1982.

[61] EETimes. "2019 Embedded Markets Study". In: (2019).

[62] eLinux. *Website of: Embedded Linux Distributions (wiki)*. https://elinux.org/Embedded_Linux_Distributions. [Online; accessed October 10, 2022]. 2022.

[63] embench. *Website of: embench-iot*. https://github.com/embench/embench-iot/tree/master/src/. [Online; accessed March 22, 2022]. 2022.

[64] Dawson R. Engler and M. Frans Kaashoek. "Exterminate all OS Abstractions". In: *HotOS* (1995).

[65] Dawson R. Engler, M. Frans Kaashoek, and James Jr. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Journal of Number Theory* 29.5 (1995), pp. 251–266. DOI: 10.1145/224057.224076.

[66] European Parliament. *Website of: Russia's war on Ukraine: Timeline of cyber-attacks*. https://www.europarl.europa.eu/RegData/etudes/BRIE/2022/733549/EPRS_BRI(2022)733549_EN.pdf. [Online; accessed October 10, 2022]. 2022.

[67] European Parliament and the Council of the European Union. *Directive on certain aspects concerning contracts for the supply of digital content and digital services*. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A32019L0770. [Online; accessed October 10, 2022]. 2022.

[68] European Telecommunications Standards Institute (ETSI). *ETSI EN 303 645: Cyber Security for Consumer Internet of Things: Baseline Requirements*. https://www.etsi.org/deliver/etsi_en/303600_303699/303645/02.01.01_60/en_303645v020101p.pdf. [Online; accessed October 10, 2022]. 2022.

[69] European Union. *EU Cyber Resilience Act*. https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act. [Online; accessed October 10, 2022]. 2022.

[70] FiCloud. *Website of: FiCloud 2021 conference*. http://www.ficloud.org/2021/. [Online; accessed October 21, 2022]. 2022.

[71] Eclipse Foundation. "IoT Developer Survey 2019 Results". In: April (2019). URL: https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf?sc_icampaign=pac_iot-developer-report&sc_ichannel=ha&sc_icontent=awssm-2530&sc_iplace=banner&trk=ha_awssm-2530.

[72] Free and Open Source Silicon Foundation. *Website of: Embench*. https://github.com/embench/embench-iot/. [Online; accessed March 18, 2022]. 2022.

[73] FreeRTOS. *Website of: FreeRTOS*. https://www.freertos.org/index.html. [Online; accessed November 20, 2020]. 2020.

[74] FreeRTOS. *Website of: FreeRTOS-MPU (FreeRTOS)*. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html. [Online; accessed November 20, 2020]. 2020.

[75] Futura. *Website of: 'Ma thèse en 3 minutes' : des doctorants vont vulgariser leurs thèses pour les rendre accessibles au plus grand nombre ! (Futura-Sciences)*. https://www.futura-sciences.com/tech/actualites/technologie-ma-these-3-minutes-doctorants-vont-vulgariser-leurs-theses-rendre-accessibles-plus-grand-nombre-99794/. [Online; accessed October 21, 2022]. 2022.

[76] Gartner. *Top Strategic IoT Trends and Technologies Through 2023*. https://www.gartner.com/en/documents/3890506-top-strategic-iot-trends-and-technologies-through-2023. [Online; January 17, 2020]. 2018.

[77] Konstantinos M Giannoutakis et al. "Next Generation Cloud Architectures". In: *The Cloud-to-Thing Continuum* (2020), p. 23.

[78] Google. *Website of: Android*. https://www.android.com/intl/fr_fr/. [Online; accessed October 10, 2022]. 2022.

[79] Google. *Website of: Fuchsia*. https://fuchsia.dev. [Online; accessed October 10, 2022]. 2022.

[80] Google. *Website of: Google Trends*. https://trends.google.com/trends/. [Online; accessed October 10, 2022]. 2022.

[81] Google. *Website of: Project Sparrow: KataOS (Github)*. https://github.com/AmbiML/sparrow-kata. [Online; accessed October 31, 2022]. 2022.

[82] Joseph Gravellier. "Remote Hardware Attacks on Connected Devices". Theses. Ecole des Mines de Saint-Etienne, Dec. 2021. URL: https://hal.archives-ouvertes.fr/tel-03491671.

[83] Michele Grisafi et al. "PISTIS: Trusted Computing Architecture for Low-end Embedded Systems". In: *USENIX - USENIX Security Symposium* (2022).

[84] Ronghui Gu et al. "CertiKOS : An Extensible Architecture for Building Certified Concurrent OS Kernels This paper is included in the Proceedings of the". In: *12th USENIX Symposium on Operating Systems Design and Implementation* (2016), pp. 653–669.

[85] Ronghui Gu et al. "CertiKOS: An extensible architecture for building certified concurrent OS kernels". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016* 4 (2016), pp. 653–669.

[86] Ronghui Gu et al. “Deep specifications and certified abstraction layer”. In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 595–608. ISSN: 15232867. DOI: 10.1145/2676726.2676975.

[87] Guillaume Bouffard and Léo Gaspard. *Website of: Choupi OS (Github)*. https://github.com/gavz/choupi-os/. [Online; accessed November 20, 2020]. 2020.

[88] Jonas Haglund and Roberto Guanciale. “Trustworthy Isolation of DMA Enabled Devices”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11952 LNCS (2019), pp. 35–55. ISSN: 16113349. DOI: 10.1007/978-3-030-36945-3_3.

[89] Gernot Heiser. “Secure embedded systems need microkernels”. In: *USENIX; login* 30.6 (2005), pp. 9–13. URL: http://www.usenix.org/publications/login/2005-12/pdfs/heiser.pdf.

[90] C. A.R. Hoare. *An axiomatic basis for computer programming*. 1969. DOI: 10.1145/363235.363259.

[91] Tony Hoare and Jay Misra. “Verified Software : Theories , Tools , Experiments Vision of a Grand Challenge Project”. In: (2008), pp. 1–18.

[92] IETF. *Website of : IETF SUIT draft architecture*. https://tools.ietf.org/html/draft-ietf-suit-architecture. [Online; accessed January 17, 2020]. 2020.

[93] IETF. *Website of : Terminology for Constrained-Node Networks (IETF)*. https://datatracker.ietf.org/doc/html/rfc7228#page-8. [Online; accessed March 22, 2022]. 2022.

[94] Alexei Iliasov, Paulius Stankaitis, and Alexander Romanovsky B. “Proving Event-B Models with Reusable Generic Lemmas”. In: (2016), pp. 210–225. DOI: 10.1007/978-3-319-47846-3.

[95] INRIA. *Website of : Coq*. https://coq.inria.fr. [Online; accessed January 17, 2020].

[96] INSPIRE-5G and AI@EDGE. *Website of: Joint INSPIRE-5G and AI@EDGE workshop – Platforms and Mathematical Optimization for Secure and Resilient Future Networks*. https://aiedge-inspire5g.roc.cnam.fr/. [Online; accessed October 21, 2022]. 2022.

[97] IoTE. *Website of: IoTE 2022 conference*. https://cndc2022.org/iote/index. [Online; accessed October 21, 2022]. 2022.

[98] Bart Jacobs, Hans Meijer, and Erik Poll. “VerifiCard: A European Project for Smart Card Verification”. In: 2001.

[99] Narjes Jomaa. “Le co-design d’un noyau de système d’exploitation et de sa preuve formelle d’isolation”. PhD thesis. 2018.

[100] Narjes Jomaa, David Nowak, and Paolo Torrini. “Formal Development of the Pip Protokernel”. In: (2018).

[101] Narjes Jomaa et al. “Formal Proof of Dynamic Memory Isolation Based on MMU”. In: *Proceedings - 10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016* (2016), pp. 73–80. DOI: 10.1109/TASE.2016.28.

[102] Narjes Jomaa et al. “Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base”. In: 076 (2018).

[103] Jonathan Salwan. *Website of: ROP gadget*. https://github.com/JonathanSalwan/ROPgadget/. [Online; accessed March 21, 2022]. 2022.

[104] Kellen Beck. *Website of: Hackers exploit casino’s smart thermometer to steal database info (Mashable)*. https://mashable.com/2018/04/15/casino-smart-thermometer-hacked/?europe=true. [Online; accessed December 10, 2020]. 2018.

[105] Chung Hwan Kim et al. “Securing Real-Time Microcontroller Systems through Customized Memory View Switching”. In: *NDSS* (2018). DOI: 10.14722/ndss.2018.23107.

[106] Gerwin Klein, Ralf Huuck, and Bastian Schlich. “Operating system verification - An overview”. In: *Journal of Automated Reasoning* 42.2-4 (2009), pp. 123–124. ISSN: 01687433. DOI: 10.1007/s10817-009-9126-9.

[107] Gerwin Klein et al. “Comprehensive Formal Verification of an OS Microkernel”. In: 32.1 (2014).

[108] Gerwin Klein et al. “Provably trustworthy systems Subject Areas :” in: *Phil.Trans.R. Soc. A* (2017). DOI: http://dx.doi.org/10.1098/rsta.2015.0404.

[109] Gerwin Klein et al. “SeL4: Formal verification of an OS kernel”. In: *SOSP’09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), pp. 207–220. DOI: 10.1145/1629575.1629596.

[110] Paul Kocher et al. “Spectre attacks: Exploiting Speculative Execution”. In: *Communications of the ACM* 63.7 (2020), pp. 93–101. ISSN: 15577317. DOI: 10.1145/3399742. arXiv: 1801.01203.

[111] Patrick Koeberl et al. “TrustLite: A security architecture for tiny embedded devices”. In: *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014* (2014). DOI: 10.1145/2592798.2592824.

[112] Krebs on Security. *Website of: DDOS on Dyn impacts Twitter, Spotify, Reddit*. https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/. [Online; accessed March 27, 2020]. 2016.

[113] Hugo Lefeuvre et al. “FlexOS”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. New York, NY, USA: ACM, 2021, pp. 79–87. ISBN: 9781450384384. DOI: 10.1145/3458336.3465292. URL: https://dl.acm.org/doi/10.1145/3458336.3465292.

[114] Stephane Lescuyer. “ProvenCore : Towards a Verified Isolation Micro-Kernel”. In: January (2015).

[115] Amit Levy et al. “Multiprogramming a 64 kB Computer Safely and Efficiently”. In: *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles* (2017), pp. 234–251. DOI: 10.1145/3132747.3132786.

[116] J. Liedtke. “Towards real microkernels”. In: *Proceedings of the 13th International Conference on Information Security Theory and Practice*. Paris, France: CACM, 1996.

[117] Linaro. *Website of: TrustedFirmware-A*. https://www.trustedfirmware.org/projects/tf-a/. [Online; accessed October 10, 2022]. 2022.

[118] Linaro. *Website of: TrustedFirmware-M*. https://www.trustedfirmware.org/projects/tf-m/. [Online; accessed November 20, 2020]. 2020.

[119] Linux Kernel Organization. *Website of: The Linux Kernel Archives*. https://www.kernel.org/. [Online; accessed October 10, 2022]. 2022.

[120] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *Communications of the ACM* 63.6 (2020), pp. 46–56. ISSN: 15577317. DOI: 10.1145/3357033.

[121] Martin Hron. *Website of: The Fresh Smell of ransomed coffee (avast.io))*. https://decoded.avast.io/martinhron/the-fresh-smell-of-ransomed-coffee/. [Online; accessed November 20, 2020]. 2020.

[122] Daniel Matichuk et al. "Empirical study towards a leading indicator for cost of formal software verification". In: *Proceedings - International Conference on Software Engineering* 1 (2015), pp. 722–732. ISSN: 02705257. DOI: 10.1109/ICSE.2015.85.

[123] Milosch Meriac. "Practical real-time operating system security for the masses (ARM TechCon 2016)". In: October (2016).

[124] Michele Andreoli. *Website of : Linux statistics*. http://micheleandreoli.org/public/Software/mulinux/. [Online; accessed January 17, 2020]. 2020.

[125] Microsoft. *Website of: An overview of Windows 10 IoT*. https://learn.microsoft.com/en-us/windows/iot-core/windows-iot. [Online; accessed October 10, 2022]. 2022.

[126] Microsoft. *Website of: Windows release health*. https://learn.microsoft.com/en-us/windows/release-health/. [Online; accessed October 10, 2022]. 2022.

[127] Charlie Miller and Chris Valasek. "Remote Exploitation of an Unaltered Passenger Vehicle". In: *Defcon 23* 2015 (2015), pp. 1–91. URL: http://illmatics.com/Remote%20Car%20Hacking.pdf.

[128] Ministère des Armées (France). *Website of: Le commandement de la cyberdéfense (COMCYBER)*. https://www.defense.gouv.fr/ema/commandement-cyberdefense-comcyber. [Online; accessed October 10, 2022]. 2022.

[129] National Institute of Standards and Technology (NIST). *IoT Cybersecurity Improvement Act of 2020*. https://www.congress.gov/bill/116th-congress/house-bill/1668. [Online; accessed October 10, 2022]. 2022.

[130] Nordic Semiconductor. *Website of: nRF52832 Product Specification v1.8*. https://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.8.pdf. [Online; accessed October 10, 2022]. 2022.

[131] Nordic Semiconductor. *Website of: nRF52840 DK (Nordic Semiconductor)*. https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk/. [Online; accessed March 18, 2022]. 2022.

[132] Nordic Semiconductor. *Website of: Power Profiler Kit (PPK)*. https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit/. [Online; accessed March 18, 2022]. 2022.

[133] Openhub. *Website of : Linux statistics*. https://www.openhub.net/p/linux. [Online; accessed January 17, 2020]. 2020.

[134] Orange. *Website of: 'Ma thèse en 3 minutes' : Grand Finale replay*. https://mastermedia.dam-broadcast.com/pm_1_96_96582-gc90egl7rp-480.mp4. [Online; accessed October 21, 2022]. 2022.

[135] Orange. *Website of: Orange*. https://www.orange.com/en. [Online; accessed October 10, 2022]. 2022.

[136] OSDev Wiki. *Website of: Exokernel*. https://wiki.osdev.org/Exokernel. [Online; accessed October 21, 2022]. 2022.

[137] OWASP. *Internet of Things (IoT) Top 10 2018*. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10. [Online; accessed October 10, 2022]. 2022.

[138] Philip Oltermann. *Website of: German parents told to destroy doll that can spy on children (The Guardian))*. https://www.theguardian.com/world/2017/feb/17/german-parents-told-to-destroy-my-friend-cayla-doll-spy-on-children. [Online; accessed November 20, 2020]. 2017.

[139] Sandro Pinto and Cesare Garlati. "Multi Zone Security for Arm Cortex-M Devices". In: ().

[140] Sandro Pinto and Cesare Garlati. "Secure IoT Firmware For Cortex-M Processors". In: ().

[141] Sandro Pinto and Nuno Santos. "Demystifying arm trustzone: A comprehensive survey". In: *ACM Computing Surveys* 51.6 (2019). ISSN: 15577341. DOI: 10.1145/3291047.

[142] Pip Club. *Website of: Pip Club*. https://listes.univ-lille.fr/sympa/info/pip-club?ticket=ST-37767111-fLzd9L4HT13NyvMxNrCsKc-pPhI-cas60-1.univ-lille.fr. [Online; accessed October 10, 2022]. 2022.

[143] Pip Development Team. *Website of: Pip*. https://pip.univ-lille.fr/. [Online; accessed October 10, 2022]. 2022.

[144] Pip Development Team. *Website of: Pip (MMU) Gitlab*. https://gitlab.univ-lille.fr/2xs/pip/pipcore/-/tree/main/. [Online; accessed October 10, 2022]. 2022.

[145] Ratish J Punnoose et al. "Survey of Existing Tools for Formal Verification." In: *Sandia Report SAND2014-20533, Sandia National Laboratories, Albuquerque* December (2014). URL: http://www.osti.gov/servlets/purl/1166644/.

[146] P. Radanliev et al. "Economic impact of IoT cyber risk - analysing past and present to predict the future developments in IoT risk analysis and IoT cyber insurance". In: *Living in the Internet of Things: Cybersecurity of the IoT - 2018* (2018), 3 (9 pp.)–3 (9 pp.) DOI: 10.1049/cp.2018.0003.

[147] Alastair Reid. "Who guards the guards? formal validation of the Arm v8-m architecture specification". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–24. ISSN: 2475-1421. DOI: 10.1145/3133912.

[148] RESSI. *Website of: RESSI 2020 conference*. http://ressi.fr/. [Online; accessed October 21, 2022]. 2022.

[149] Richard Stallman. *Website of: GNU/Linux FAQ*. https://www.gnu.org/gnu/gnu-linux-faq.html. [Online; accessed October 21, 2022]. 2022.

[150] Talia Ringer. "Proof Repair". PhD thesis. University of Washington, 2021, pp. 1–162.

[151] RIOT Development Team. *Website of: RIOT OS (Github)*. https://github.com/RIOT-OS/RIOT. [Online; accessed October 10, 2022]. 2022.

[152] RISC-V. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (Document Version 20211203)*. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf. [Online; accessed October 10, 2022]. 2022.

[153] Ryan Roemer et al. "Return-oriented programming: Systems, languages, and applications". In: *ACM Transactions on Information and System Security* 15.1 (2012), pp. 1–34. ISSN: 10949224. DOI: 10.1145/2133375.2133377.

[154] J M Rushby. "Proof of Separability: A Verification Technique for a Class of Security Kernels". In: (1984), pp. 352–367.

[155] J. M. Rushby. "Design and verification of secure systems". In: *Proceedings of the 8th ACM Symposium on Operating Systems Principles, SOSP 1981* 15.5 (1981), pp. 12–21. DOI: 10.1145/800216.806586.

[156] John Rushby. "Partitioning in avionics architectures: requirements, mechanisms and assurance". In: *Work* March (2000), p. 67. DOI: DOT/FAA/AR-99/58.

[157] Rust. *Website of: The Rust programming language (Rust)*. https://www.rust-lang.org/. [Online; accessed March 27, 2020]. 2020.

[158] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted execution environment: What it is, and what it is not". In: *Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015* 1 (2015), pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[159] Jerome H. Saltzer and Michael D. Schroeder. "The protection of of information computer systems". In: 63 (1975), pp. 1278–1308.

[160] Sean Michael Kerner. *Website of: Industroyer Cyber-Attack Revealed as Cause of Ukraine Power Outage (Eweek)*. https://www.eweek.com/security/industroyer-cyber-attack-revealed-as-cause-of-ukraine-power-outage. [Online; accessed March 27, 2020]. 2017.

[161] SEAS. *Website of: SEAS group (CRIStAL laboratory)*. https://www.cristal.univ-lille.fr/en/gt/seas/. [Online; accessed October 10, 2022]. 2022.

[162] seL4. *Website of: FAQ seL4 including DMA*. https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html. [Online; accessed October 10, 2022]. 2022.

[163] Abderrahmane Sensaoui et al. "An In-depth Study of MPU-Based Isolation Techniques". In: *Journal of Hardware and Systems Security* 3.4 (2019), pp. 365–381. ISSN: 2509-3428. DOI: 10.1007/s41635-019-00078-6.

[164] Rui Shu et al. "A study of security isolation techniques". In: *ACM Computing Surveys* 49.3 (2016). ISSN: 15577341. DOI: 10.1145/2988545.

[165] Miguel Silva et al. "Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking". In: *IEEE Internet of Things Journal* 6.6 (2019), pp. 10375–10383. ISSN: 23274662. DOI: 10.1109/JIOT.2019.2939008.

[166] Philip Sparks. "The route to a trillion devices The outlook for IoT investment to 2035". In: *ARM Whitepaper* (2017), pp. 1–14.

[167] Mark Staples et al. "Productivity for Proof Engineering". In: (2014), pp. 1–4.

[168] Statista. *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030*. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. [Online; October 25, 2022]. 2022.

[169] Editors Martin R Stytz and James A Whittaker. "Considering defense in depth for software applications". In: ().

[170] Gang Tan. "Principles and Implementation Techniques of Software-Based Fault Isolation". In: *Principles and Implementation Techniques of Software-Based Fault Isolation* (2017). DOI: 10.1561/9781680833454.

[171] Tanenbaum. *Website of: MINIX 3*. https://www.minix3.org/. [Online; accessed November 20, 2020]. 2020.

[172] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. USA: Prentice-Hall, Inc., 2005. ISBN: 0131429388.

[173] Trustworthy Systems Team. "seL4 Reference Manual Version 3.2.0". In: July (2016).

[174] TinyPART members. *Website of: TinyPART*. https://TinyPART.github.io/TinyPART. [Online; accessed October 10, 2022]. 2022.

[175] Tock development team. *Website of: Tock*. https://www.tockos.org/. [Online; accessed November 20, 2020]. 2020.

[176] TockOS. *Website of: Tracking: Support RISC-V (Github)*. https://github.com/tock/tock/issues/1135. [Online; accessed October 10, 2022]. 2022.

[177] University of Cambridge and Technische Universität München. *Isabelle*. https://isabelle.in.tum.de/index.html. [Online; accessed October 10, 2022]. 2022.

[178] Florian Vanhems et al. "On the Proof-Oriented Design of a Context-Switching Service in the Pip Protokernel". In: *ENTROPY 2019* (2019).

[179] Arun Viswanathan and B C Neuman. "A survey of isolation techniques". In: *Information Sciences Institute, University of Southern California* (2009), pp. 1–16.

[180] Robert Wahbe et al. "Efficient software-based fault isolation". In: *SOSP 1993 - Proceedings of the 14th ACM Symposium on Operating Systems Principles* (1993), pp. 203–216. DOI: 10.1145/168619.168635.

[181] Robert N M Watson et al. "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture". In: 891 (2014). ISSN: 1476-2986. URL: http://www.cl.cam.ac.uk/.

[182] Wikipedia. *IBM System360 Model 40*. https://en.wikipedia.org/wiki/IBM_System/360_Model_40. [Online; accessed October 10, 2022]. 2022.

[183] Wikipedia. *Website of: List of Linux Distributions (wiki)*. https://en.wikipedia.org/wiki/List_of_Linux_distributionss. [Online; accessed October 10, 2022]. 2022.

[184] Wikipedia. *Website of: Stuxnet (Wikipedia)*. https://en.wikipedia.org/wiki/Stuxnet. [Online; accessed March 27, 2020]. 2020.

[185] Wilgaard. *Website of: ppk_api (Github)*. https://github.com/wlgrd/ppk_api. [Online; accessed March 23, 2022]. 2022.

[186] Xavier Leroy. *Website of: CompCert*. https://compcert.org/. [Online; accessed November 20, 2020]. 2020.

[187] Hongyan Xia et al. "CheriRTOS: A Capability Model for Embedded Devices". In: *Proceedings - 2018 IEEE 36th International Conference on Computer Design, ICCD 2018* (2019), pp. 92–99. DOI: 10.1109/ICCD.2018.00023.

[188] Shenghao Yuan et al. "End-to-End Mechanized Proof of an eBP Virtual Machine for Micro-controllers". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 13372 LNCS (2022), pp. 293–316. ISSN: 16113349. DOI: 10.1007/978-3-031-13188-2_15.

[189] Zephyr Project. *Website of: The Zephyr Project*. https://www.zephyrproject.org/. [Online; accessed November 20, 2020]. 2020.

[190] Zephyr RTOS development team. *Website of: Memory Management for MMU-based devices for LTS2 (Github))*. https://github.com/zephyrproject-rtos/zephyr/issues/27819. [Online; accessed October 25, 2022]. 2020.

[191] Zephyr RTOS development team. *Website of: Memory Protection Design (Github))*. https://github.com/zephyrproject-rtos/zephyr/blob/d048faacf234061f6fb70e4926446268a68442bd/doc/reference/usermode/memory_domain.rst. [Online; accessed October 25, 2022]. 2019.

[192] Yongwang Zhao. "A survey on formal specification and verification of separation kernels". In: 11.4 (2016), pp. 1–23.

[193] Yongwang Zhao et al. "High-Assurance Separation Kernels: A Survey on Formal Methods". In: 1.1 (2017). arXiv: 1701.01535. URL: http://arxiv.org/abs/1701.01535.

[194] Wei Zhou et al. "Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems". In: *arXiv preprint arXiv:1908.03638* (2019). arXiv: 1908.03638. URL: http://arxiv.org/abs/1908.03638.

[195] Xia Zhou et al. "OPEC: operation-based security isolation for bare-metal embedded systems". In: *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5-8, 2022, RENNES, France* 1.1 (2022), pp. 317–333. DOI: 10.1145/3492321.3519573.

Appendix A

Résumé substantiel en français

Chapter outline

La motivation principale des travaux présentés dans ce manuscrit provient du problème de sécurité dans les systèmes informatiques contraints en ressources (mémoire, puissance, énergie). Avec une sécurité faible et de plus en plus connectés, ces appareils sont la cible privilégiée de cybercriminels qui peuvent en extraire des données sensibles ou les utiliser comme points d'entrée dans des réseaux protégés. Ces objets contraints se trouvent sous toutes les formes (ampoule, détecteur de fumée, thermomètres,...), dans tous les milieux (domiciles, usines, entreprises...), et se connectent à l'Internet des Objets (Internet of Things (IoT), en anglais) pour servir des écosystèmes de plus en plus intelligents (jumeaux numériques, IoT-Cloud Continuum [77]...). En étant connecté, ces objets rapprochent les attaquants des systèmes privés protégés.

La première intuition lors d'une attaque, même en-dehors du monde informatique, est l'isolation : s'isoler de l'attaquant ou isoler l'attaquant. C'est aussi une solution pour répondre aux vulnérabilités mémoire qui peuvent être exploitées par ces attaquants. Cette propriété est *nécessaire* pour garantir l'état courant où d'autres propriétés pourraient être vérifiées. Cependant, cette solution n'est *pas suffisante* pour la sécurité, par exemple un appareil qui effectue une mise-à-jour avec un microgiciel corrompu se protégera par d'autres mécanismes comme la signature électronique ou l'authentification. Par ailleurs, l'isolation est fondamentale pour d'autres principes de sécurité tels que MILS (Multiple Independent Levels of Security/Safety) et TEE (Trusted Execution Environments) [24, 158].

Cette thèse est réalisée conjointement entre Orange et l'Université de Lille (thèse CIFRE). Elle a démarrée le 28 novembre 2019. Orange [135], en tant qu'opérateur, est intéressé par sécuriser les appareils de son réseau pour réduire les communications malicieuses. Les appareils piratés pourraient prendre une part importante de la bande passante et impacter la qualité de service d'appareils légitimes, mais peuvent aussi attaquer les propres infrastructures d'Orange ou bien prendre le contrôle d'appareils connectés pour réaliser des cyber-attaques massives à des fins criminelles ou terroristes. Orange soutient ses travaux en publiant l'ensemble des résultats en licence ouverte. L'équipe 2XS [1] de L'Université de Lille à laquelle je suis rattaché est spécialiste de la sécurité des objets fortement contraints en ressources. L'objectif de la thèse a été l'adaptation du noyau Pip développé par 2XS pour l'environnement des objets contraints. Ce nouveau projet, appelé Pip-MPU, prend son nom du projet parent Pip et d'un

composant informatique de protection mémoire nommé MPU. Pip-MPU est partie prenante du projet TinyPART [174], un consortium de partenaires académiques et industriels franco-allemand dont incluant Orange et l'Université de Lille.

A.0.1 Questions de recherche

Pour contrer les menaces pesant sur les économies et les cyber-attaques massives ayant eu lieu ces dernières années [184, 160], les marchés des consommateurs et industriels reçoivent désormais des obligations réglementaires et des standards qui concernent les objets contraints [69, 68]. Les appareils vérolés sont utilisés comme vecteurs d'attaque et d'intrusion (extraction de données client d'un casino par le piratage du thermomètre connecté de son aquarium [104]) ou comme armes (sites web bloqués par l'attaque massive Mirai [112], rançongiciel chargé sur ceinture de chasteté connectée [27]).

La thèse explore les questions suivantes :

- Pourquoi est-ce que la sécurité est si peu prioritaire pour les développeurs de logiciels embarqués et par transitivité aux systèmes qu'ils développent ?
- Quelle facilité y a-t-il pour les attaquants à distance de détourner les appareils embarqués pour les transformer en armes utilisées lors de cyber-attaques massives ?
- Comment les objets contraints peuvent-ils se prémunir contre ces attaques ? Ont-ils les briques technologiques nécessaires et adaptése à l'environnement contraint pour se protéger ?
- Quels types d'attaques peuvent-ils repoussés avec ces briques de sécurité ?
- Quel est le niveau de confiance attendu pour des appareils contraints protégés ?
- Est-ce que des solutions génériques et portables existent couvrant la majorité de ces appareils malgré leur hétérogénéité ?

Pour répondre à ces questions, nous nous intéressons à l'isolation pour la sécurité des objets contraints, en particulier les solutions d'isolation ancrées dans le matériel. Nous recherchons des solutions applicables au niveau du système, tels que des noyaux, des outils ou des composants matériels. Les solutions sur cible nue ne nous intéressent pas car non universelles.

A.0.2 Modèle d'attaquant

Notre attaquant cherche à prendre le contrôle total de l'appareil, à distance, et peut installer ses propres composants logiciels. Les attaques logicielles sont particulièrement menaçantes pour les systèmes par l'exploitation de vulnérabilités mémoire dues à des erreurs de programmation ou des dépassements de tampons, des accès mémoire illégaux, des corruptions de données. Nous considérons que l'état du système initial est bénin mais entaché de ces vulnérabilités. L'exploitation en elle-même peut se faire par injection de code, techniques de réutilisation de code, opérations illégales de lecture et d'écriture [105, 32, 111, 52]. L'attaquant prend le contrôle total par l'intermédiaire d'un composant privilégié ou en réussissant à s'octroyer des droits privilégiés.

A.0.3 Eléments préliminaires

Ce résumé est à destination de personne ayant déjà certaines connaissances en informatique poussées. Nous détaillons ici certaines notions avancées, non généralistes et propres à ces travaux de thèse tandis que le reste est considéré acquis.

Tout d'abord, les travaux de thèse sont destinés aux systèmes embarqués. Ceux-ci dépendent fortement des facteurs SWaP-C (taille, poids, puissance et coûts). Ils peuvent néanmoins être très hétérogènes, par exemple une voiture embarque des centaines de micro-contrôleurs haute-performance à mettre en opposition au microcontrôleur peu puissant du grille-pain. Nous nous intéressons particulièrement dans cette thèse aux objets fortement contraints, de classe 2 dans le classement IETF reporté dans le Tableau A.1, où les facteurs SWaP-C sont encore plus importants que d'autres. Nous constatons actuellement que les systèmes embarqués préalablement isolés (télécommande, détecteur de fumée, thermomètre...) sont de plus en plus connectés, et ce mouvement s'accélère selon les études [61, 71]. Cependant, il est aussi observé par lecture croisée de ces études que les systèmes antérieurs n'adoptent pas encore les standards de sécurité requis dans le monde connecté, alors même que nous recevons des alertes de chercheurs en sécurité (piratage d'une Jeep Cherokee à distance permettant la prise de contrôle du volant et des freins [127], rançongiciel sur une machine à café [121]) et que nous sommes témoins de cyber-attaques massives (botnet Mirai [112]) et éventuellement dévastatrices (ver informatique STUXNET développé notamment par la NSA qui s'est introduit dans des centrifugeuses nucléaires iraniennes pour saboter le programme nucléaire iranien [184], attaques sur le réseau électrique ukrainien en 2015 et 2016 [160, 66]).

Name	Taille données (*e.g.*, RAM)	Taille code (*e.g.*, Flash)
Classe 0, C0	< 10 Kio	< 100 Kio
Classe 1, C1	10 Kio	100 Kio
Classe 2, C2	50 KiBo	250 Kio

Table A.1: Classes d'objets contraints (Kio = 1024 octets) [93]

Pour un accès contrôlé à la mémoire, les systèmes généralistes reposent sur la MMU (Memory Management Unit). La MMU met en place une virtualisation mémoire sous forme de pagination et attribue à chaque page des droits d'accès. Son équivalent dans les objets contraints est l'unité *facultative* appelée Memory Protection Unit (MPU). Celle-ci est aussi reconfigurable pendant l'exécution par des composants logiciels privilégiés et de façon équivalente aux pages, protège des régions mémoire selon des permissions d'accès définies par l'utilisateur. Una accès mémoire illégitime, aussi bien dans la MMU que la MPU, se termine en faute mémoire. Néanmoins, ces deux unités sont aussi très différentes. Le stockage de leur configuration se fait en mémoire principale pour la MMU mais dans des registres pour la MPU. En effet, la MPU est beaucoup plus limitée et ne propose que 16 régions mémoire protégées en même temps, là où la MMU propose des millions de pages. Par ailleurs, les régions MPU sont de taille variable alors que les pages MMU sont le plus souvent de taille fixe. ces différences sont reportées dans le Tableau A.2 et sont autant de difficultés structurelles pour adapter un système basé sur une unité à l'autre. La MPU est liée aux architectures ARM mais dispose d'équivalents dans d'autres architectures de processeurs tels que la Physical Memory Protection (PMP)

Attributs	MMU	MPU
Mémoire virtuelle	Oui	Non
Mode de configuration	privilégié	privilégié
Unité de région mémoire	page	région MPU
Nombre d'unités de région mémoire	Millions	8-16
Contrôle d'accès (RWX)	Oui	Oui
Stockage de la configuration	mémoire principale	registre
Taille mémoire des appareils	MB-GB	kB
Fréquence CPU des appareils	GHz	MHz

Table A.2: MMU *versus* MPU.

de RISC-V [152].

Pour finir, nous nous intéressons particulièrement aux proto-noyaux, une famille de systèmes d'exploitation illustrée dans la Figure A.1. Les communautés concevant les systèmes d'exploitation (SE, ou Operating System (OS) en anglais) voient depuis longtemps défiler de nombreuses définitions, notamment celles données par Andrew Tanenbaum [4] et Richard Stallman [149] qui a une vision plus large, et souvent mélangées à la définition du noyau du SE. Nous définissons le système d'exploitation comme l'abstraction des ressources matérielles et son noyau comme tous les sous-composants qui contiennent les fonctionnalités critiques et qui s'exécutent en mode noyau (mode privilégié). Ce qui est ou non considéré comme critique dépend de la philosophie de chaque noyau. Nous distinguons habituellement les noyaux monolithiques (comme Linux [119], contenant l'ensemble du systèmes d'exploitation dans l'espace noyau) des micro-noyaux (seuls les fonctionnalités de gestion du matériel, d'ordonnancement, de multiplexage et de communications inter-processus sont gardées dans l'espace noyau, le reste est rejeté dans l'espace utilisateur, comme pour Minix [171]). D'un autre point de vue, les exo-noyaux rejettent toutes les abstractions pour se focaliser sur la sécurité et, encore plus drastiquement, les proto-noyaux comme Pip [143], qui ne proposent que des services de gestion mémoire et de flot de contrôle.

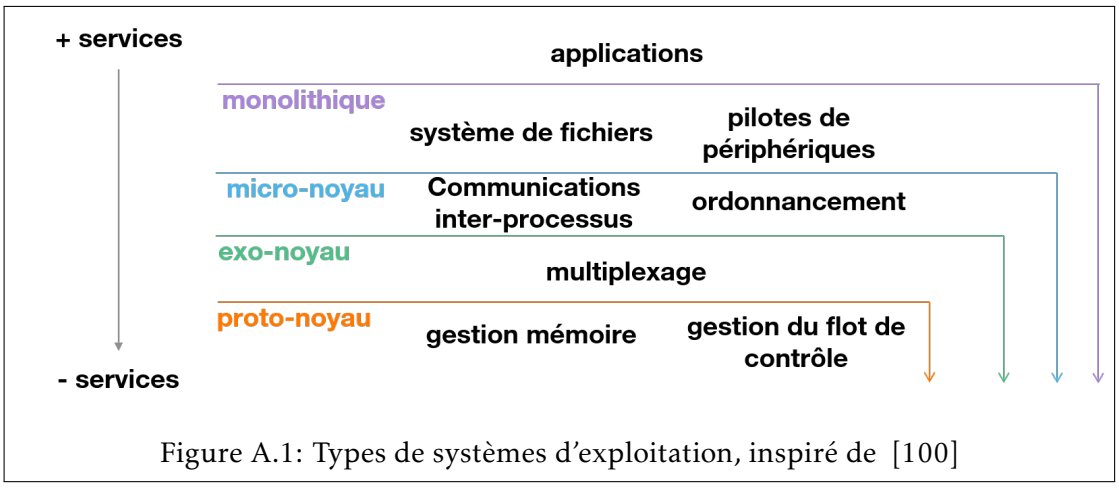


Figure A.1: Types de systèmes d'exploitation, inspiré de [100]

A.0.4 Etat de l'art

Nous nous focalisons sur les solutions d'isolation ancrées dans le matériel pour les systèmes embarqués.

Dans le secteur des petits objets connectés, les garanties d'isolation sont faibles, voire n'existent pas.

Les systèmes les plus connus (RIOT OS, FreeRTOS, Zephyr, TinyOS, Contiki, MbedOS [165, 71]) proposent un portage sur des systèmes dotés de MPU pour mettre en place une isolation, mais ces portages sont bien souvent soit inefficaces, soit non utilisés, soit non maintenus. Par exemple, dans son portage MPU, RIOT-OS [22] met en place des garde-fous pour les piles des fils d'exécution et rend inexécutable la RAM, mais laisse le reste des données et code des fils d'exécution et le système d'exploitation ouverts aux modifications d'un composant malveillant ou maladroit. FreeRTOS-MPU [74], la variante de FreeRTOS [73] avec utilisation de la MPU, propose d'utiliser la MPU pour protéger le noyau des applications non privilégiées et limite l'accès mémoire de certaines tâches à leur pile d'exécution et à trois zones supplémentaires arbitrairement fixées par l'utilisateur. La MPU limite aussi les droits d'accès au code des tâches en lecture et exécution seulement (pas d'écriture). Cette utilisation permet de détecter des débordements de pile ou l'accès à des zones non autorisées et limite les injections de code, tout en protégeant constamment le noyau. Mais la variante propose aussi de créer des tâches avec accès total à toute la RAM, exposant les données des autres tâches, ou encore permet de configurer les trois zones utilisateurs pour avoir accès aux données des autres tâches en tant que mémoire partagée, ou vol/corruption de données dans le cas frauduleux. Par ailleurs, le code d'une tâche est tout le temps accessible par les autres. Et finalement, il est possible de créer des tâches privilégiées, qui aurait les mêmes droits que le noyau et pourraient dans le cas malicieux le supplanter. Nous avons également étudié les systèmes TrustedFirmware-M [118] et la variante basée MPU de Zephyr [189].

Les systèmes moins traditionnels ne conviennent pas non plus au problème posé pour plusieurs raisons. Dans TockOS [175], la configuration de la MPU est dédiée au processus courant et isole le code et les données en changeant de configuration à chaque changement de contexte. La MPU est cette fois complétement désactivée lorsque le noyau s'exécute. Ici, aucune région MPU ne couvre alors les parties noyau (hormis certaines structures protégées et inaccessibles pour le processus). Le principe d'isolation est donc entièrement respecté. En revanche, l'isolation n'est pas formalisée et le système ne propose qu'un seul niveau de protection, c'est-à-dire qu'il n'est pas possible de protéger des processus enfants. Des travaux ont aussi proposés une machine virtuelle JavaCard (JCVM) endurcie par la MPU [30]. Dans ce cas, lors d'un changement de contexte, la MPU modifie uniquement la région MPU dédiée à protéger la pile d'exécution comprenant le contexte de l'application courante. Les autres régions MPU sont toutes partagées avec les autres applications. Aucune possibilité de posséder des régions mémoire supplémentaires comme dans FreeRTOS-MPU. Encore une fois, les propriétés d'isolation ne sont pas prouvées et de nombreuses régions sont partagées, rompant le principe d'isolation. Dans EwoK [25], certaines régions MPU sont dédiées au noyau et fixées lors de la compilation puis protégées lors de l'exécution, les autres changent à chaque changement de contexte et protègent code, données et périphériques de l'application courante. EwoK utilise les langages Ada/SPARK pour

garantir aucune erreur lors de l'exécution et rédige des contrats qui spécifient les exigences fonctionnelles des composantes critiques, vérifiés automatiquement. Aussi, un contrat SPARK développé impose qu'aucune région MPU ne soit à la fois ouverte à l'écriture et à l'exécution (principe $W \wedge X$). Le principe d'isolation est entier. Néanmoins, les preuves d'isolation plus compliquées ne sont pas présentes, bien qu'elles puissent être réalisées en SPARK moyennant un effort non négligeable de rédaction de preuves. Nous avons étudié également les systèmes MultiZone [139, 140], et les systèmes modifiant le matériel TrustLite [111] et CheriRTOS [187].

Par ailleurs, d'autres outils et architectures traitent du problème d'isolation pour les petits systèmes. Par exemple, l'architecture MINION [105] propose une analyse du code source afin d'obtenir des vues mémoire pour chaque processus qui s'exécutera et configure la MPU à chaque changement de contexte pour refléter ces vues. Cela empêche les autres processus d'accéder à une mémoire qui ne leur est pas normalement dédiée après analyse. Mais MINION adapte ses vues mémoire au nombre de régions MPU disponibles, et de ce fait un processus se voit attribuer des zones mémoire plus larges que nécessaires, notamment des périphériques ou données d'autres processus. Cette tendance est également présente dans ACES (Automatic Compartments for Embedded Systems) [32], un outil qui instrumente le binaire pour créer des compartiments isolés. Les compartiments sont créés en fonction de la politique de compartimentation décidée par l'utilisateur, par exemple en rassemblant les composants issus du même fichier source. Cependant, la dernière étape est de réduire le nombre de compartiments au nombre de régions MPU disponibles, ce qui a pour conséquence de finalement rassembler des composants que l'on aurait voulu isoler. Une version améliorée d'ACES et de MINION est OPEC [195], cependant n'est destinée qu'aux systèmes sur cible nue.

Pour tous ces systèmes embarqués, majoritairement en code source ouvert, la confiance demandée est plus importante qu'avec des preuves formelles d'isolation. A notre connaissance, ProvenCore-M [114, 28] est le seul système d'exploitation qui propose les mêmes garanties que celles visées dans cette thèse, mais étant un système propriétaire, peu d'informations publiques existent sur sa conception et il n'est donc pas engagé dans la diffusion de la technologie au plus grand nombre.

Du côté des systèmes embarqués avec plus du puissance et plus d'espace mémoire, nous trouvons les SEs conventionnels comme les distributions Linux, Windows, MacOS and Android [183, 126, 8, 78]. Plus spécifiques à l'environnement embarqué, nous trouvons les membres de la famille Windows IoT, Linux Embedded et Fuchsia [125, 62, 79]. Ils proposent tous une isolation basée sur la mémoire virtuelle mise en place par la MMU.

Pip [143] fait partie de la famille des proto-noyaux et est spécialisé dans l'isolation mémoire. C'est un noyau de séparation, destiné à améliorer la sécurité des systèmes d'exploitation [155], par exemple comme support à une architecture Multiple Independent Levels of Security/Safety (MILS)[24]. Par ailleurs, il est open-source et donc étudiable et adaptable et participe à une démarche de sécurité ouverte permettant de valider la confiance dans le système. Pip mobilise le matériel pour asseoir ses garanties d'isolation, soit la MMU, rendant une attaque à distance dans l'objectif de rompre l'isolation difficile à réaliser. Dans ce secteur, d'autres systèmes font bonne figure, telle que le noyau seL4 [109, 173, 89] ou mCertiKOS-secure [41, 86, 84] qui se basent aussi sur la MMU et prouvent l'isolation par raffinement qui est une autre démarche de

vérification formelle depuis un modèle très abstrait jusqu'à un modèle concret qui sert à l'implémentation.

A.0.5 Problématique

Les solutions proposées de l'état de l'art présentent différentes techniques d'isolation avec des niveaux de garanties, de facilité d'utilisation et de fonctionnalités variables. Les solutions hybrides combinant ancrage matériel et isolation formellement prouvée démontrent l'isolation la plus forte.

Cependant, aucun système pour objets contraints ne propose ce niveau de garantie. En effet, la majeure partie d'entre eux ne sont pas soutenus par des méthodes formelles qui sont nécessaires pour obtenir des garanties mathématiques. Même lorsqu'ils utilisent les méthodes formelles, les solutions sont peu génériques, n'analysent pas la base de confiance entière, ont une implémentation et donc des preuves fortement liées à une architecture matérielle spécifique et sont extrêmement limités en nombre de composants protégés. De plus, certains reposent sur les propriétés intrinsèques des langages de programmation pour renforcer leur sécurité, cependant n'adresse pas l'isolation qui est plus abstraite. Par ailleurs, une isolation matérielle paraît plus forte car elle traite non plus les causes des vulnérabilités mais leurs effets. En dernier lieu, beaucoup de systèmes n'ont aucun moyen d'isolation et sont alors vulnérables à des attaques d'injection de code, d'élévation de privilèges, de vols ou de corruption de données, ou encore une corruption du système sous-jacent.

Alors que les briques technologiques existent pour mettre en place une solution de sécurité ancrée dans le matériel pour objets contraints avec de fortes garanties d'isolation, le processus de développement nécessite des experts en conception de système et en méthodes formelles. Avec un marché des objets connectés en forte hausse ces dernières années, l'industrie a été plus sensible aux ventes que de sécuriser les appareils pour maintenir un coût de production bas. Cette équation n'a pas encore trouvée de solution, laissant ouverte un manque illustré dans le Tableau A.3. Seul le système ProvenCore-M semble convenir à nos exigences, cependant il est propriétaire et peu d'informations sont divulguées à son sujet. Les systèmes hybrides (solutions logicielles ancrées dans le matériel) atteignent le niveau de garantie d'isolation désirée, cependant ils se basent sur la MMU et donc la solution n'est pas transférable telle quelle pour les objets contraints. Néanmoins, ils reposent plus précisément sur le rôle de contrôle d'accès de la MMU qui a son équivalent dans les objets contraints avec la MPU.

En résumé, nous n'avons pas trouvé dans notre état de l'art un système qui :

- est un **composant logiciel** ;
- démontre de **fortes garanties d'isolation mémoire** assurées par ;
  - **ancrage matériel** ET
  - **preuves formelles**
- est adapté aux objets contraints ;
- open-source.

	Garanties d'isolation **intermédiaire**	Garanties d'isolation **élevée** (par méthodes formelles)
Systèmes embarqués **très** contraints	TrustedFirmware-M [118], RIOT OS (MPU) [22], Zephyr RTOS (MPU) [189], FreeRTOS-MPU [74], EwoK [25], MINION [105], ACES [32], OPEC [195], MultiZone [139], TrustLite [111], CheriRTOS [187], MbedOS [20], TockOS [175], Java Card Virtual Machine endurcie [30]	ProvenCore-M [28] (propriétaire)
Systèmes embarqués **peu** contraints	Windows IoT [125], Linux Embedded [62], Fuchsia [79], TrustedFirmware-A	seL4 [89], mCertiKOS-secure [41], Pip [26, 99], ProvenCore [28] (propriétaire)

Table A.3: Solutions de sécurité ancrées dans le matériel pour systèmes embarqués et systèmes généralistes (SEs/noyaux, outils, architectures) classées par garanties d'isolation.

A.0.6 Thèse

Thèse: Les objets contraints, en particulier connectés, bénéficieraient d'une sécurité très forte atteignable avec les briques technologiques existantes tout en étant compatible avec les attentes fonctionnelles d'un écosystème dynamique. Une solution de sécurité ancrée dans le matériel, immédiatement déployable sur ces appareils, adaptée au monde réel et démontrant de fortes garanties de sécurité, peut faire bénéficier aussi bien les consommateurs que les acteurs industriels.

L'objectif est de ne jamais laisser une application interférer avec d'autres applications ou le système d'exploitation d'une manière qui pourrait mettre en danger la confidentialité ou l'intégrité de composants logiciels protégés ou pourrait permettre de prendre le contrôle total d'un système en exploitant par exemple une vulnérabilité mémoire.

Un noyau fournissant une isolation spatiale stricte mais flexible permet aux appareils d'évoluer librement dans un écosystème dynamique et empêche un attaquant à distance de prendre le contrôle total de l'appareil et s'infiltrer dans les réseaux auxquels il est connecté pour perpétrer des attaques de l'intérieur. La vérification formelle de l'isolation renforce la confiance de l'utilisateur dans le système et participe plus largement à l'adoption des écosystèmes *Internet of Things (IoT)*, qui, à leur tour dynamisent l'industrie.

L'approche suivie dans cette thèse pour sécuriser les objets contraints est la conception d'un noyau de système d'exploitation formellement vérifié qui fournit une isolation stricte ancrée dans le matériel mais flexible par l'utilisation d'une MPU prise sur étagère (*Commercial Off-the-Shelf (COTS)*, en anglais). Nous avons choisi d'adapter un noyau formellement vérifié destiné aux systèmes généralistes, en l'occurrence le proto-noyau Pip, pour l'environnement des objets contraints. Le but est de minimiser les efforts pour

maintenir un coût de conception bas. En effet, la thèse s'appuie sur l'expertise de notre équipe de recherche qui a développé et formellement vérifié Pip. De plus, l'isolation est un principe de base en sécurité informatique mais aussi indispensable pour d'autres raisons telle que la sûreté de fonctionnement, la modularité logiciel, le recouvrement de fautes, la facilitation de l'évolution d'un systèmes...[179]. Sans isolation, rien n'est moins sûr puisque les propriétés vérifiées autrement pourraient changer. C'est une première barrière lors de la mise en place d'une stratégie de défense en profondeur [169, 6]. Pip dispose d'une grande flexibilité au sein de son modèle de partitionnement. Il est aussi construit autour d'une base de confiance minimale (*Trusted Computing Base (TCB)*, en anglais) qui est bénéfique pour la sécurité en réduisant la surface d'attaque mais aussi participe à garder des coûts d'adaptation bas. Par ailleurs, Pip a déjà su montrer les résultats de son processus de preuve. Aussi, Pip évite de vérifier également l'application qui s'exécute puisque celle-ci est libre d'évoluer telle qu'elle le souhaite au sein des murs de sécurité érigés par Pip. De surcroît, cette approche propose d'utiliser la MPU qui est fortement disponible sur le marché de l'*IoT* par la domination du concepteur Arm (67% des parts de marché rien qu'avec l'architecture ARMv7 [71]). Une MPU prise sur étagère ne requiert pas non plus de modifications matérielles qui augmenteraient les coûts de conception. En héritant les caractéristiques du projet parent Pip, le noyau adapté sera également mis en source ouverte où sa sécurité pourra être évaluée par n'importe qui. Pip utilise également des langages de programmation adaptés et très utilisés par les communautés expertes du logiciel (langage C) et de vérification formelle (Gallina, le langage de spécification de l'assistant de preuve Coq). Enfin, une adaptation de Pip permet de se détacher des solutions de sécurité existantes sur MPU qui ont une flexibilité fortement limitée et par ailleurs s'appuie sur une communauté existante (le Pip User Club [142]) avec des résultats précédents positifs qui renforceraient l'acceptabilité d'une telle solution.

Cette approche permet la conception d'un noyau sécurisé dès la conception immédiatement disponible pour l'industrie et s'appuyant sur une base de confiance minimale ancrée dans le matériel. Pip adapté aux objets contraints répond aux environnements à criticité mixte, permet à plusieurs acteurs de faire tourner leurs applications de façon sécurisée sur le même appareil, avec des composants multi-processus et multi-threadés pour des applications sur cible nue ou bien s'appuyant sur un OS porté.

A.0.7 Obstacles

Nous listons ici les obstacles à la réalisation de la thèse et qui servent également de lices lors du développement. En premier lieu, aucun noyau d'isolation existant n'a adapté leur solution basée sur la MMU vers la MPU ou vice-versa. De plus, l'effort de preuve est rapporté très important dans les noyaux formellement prouvés et difficile à combiner pour un projet de thèse limité dans le temps qui inclut également une refonte de la conception lors de l'adaptation [100, 41, 109]. Par ailleurs, il n'existe à ce jour aucun système possédant la preuve formelle d'isolation par l'utilisation de la MPU, ce qui implique de la modéliser et spécifier son fonctionnement. Aussi, les solutions existantes sont extrêmement enracinées dans le matériel sous-jacent, ce qui les fait perdre en portabilité, alors que nous souhaitons une solution plus globale. Enfin, les coûts de sécurité, c'est-à-dire des performances atténuées à cause de la couche de

sécurité, peuvent rendre difficile une utilisation dans les systèmes temps-réel.

A.0.8 Résultats

Cette dissertation consolide notre thèse A.0.6 en déclinant notre approche en deux parties.

La première partie traite de la conception du noyau sécurisé pour objets contraints, nommés ci-après Pip-MPU, qui est une adaptation du noyau Pip pour les appareils disposant d'une MPU. Pip-MPU est décrit et nous montrons que sa conception satisfait les exigences de sécurité, de performance, fonctionnelles et matérielles attendues pour Pip et l'environnement contraint. Ce noyau est prototypé sur un kit de développement basé sur l'architecture ARMv7 Cortex-M doté d'une MPU. Ce prototype est évalué en termes de performances, d'empreinte mémoire, de consommation énergétique et de gains de sécurité. Pip-MPU est présenté dans les articles [55, 56] et accessible sous licence open-source[1]. Pip-MPU est une spécialisation d'un cadriciel que nous avons développé et qui permet de mettre en place une compartimentation flexible et sécurisée sur plusieurs niveaux d'abstraction. Une unique entité privilégiée dispose du contrôle de la MPU. Tous les composants logiciels s'exécutent dans l'espace utilisateur et doivent s'adresser à l'entité privilégiée pour modifier la configuration de la MPU et mettre en place une politique de sécurité locale. Le cadriciel est présenté dans les articles [54, 57].

Dans la seconde partie, nous vérifions la propriété d'isolation de Pip-MPU par l'usage de méthodes formelles. Les preuves sont d'abord réalisées informellement avant d'être conduites au sein de l'assistant de preuve Coq [95]. Nous présentons la conduite de preuve sur deux services représentatifs du noyau. Le développement des preuves est évalué en termes d'effort humain, de ressources machine et de couverture de code vérifié. Le processus de vérification formelle fait intervenir de nouvelles techniques de preuves qui sont décrites dans la thèse. Nous développons aussi de nouvelles métriques de preuve permettant la mise en place d'une stratégie globale de preuve qui minimise les efforts de preuve à chaque étape. Ces métriques vont être présentées pour la première fois lors de la présentation 'Formal Proof Metrics : the Developer's Guide to Formal Proofs' [96].

Ces résultats confirment la thèse par la démonstration de la conception d'un noyau pour objets contraints, son implémentation et sa vérification formelle qui présente de fortes garanties d'isolation mémoire.

A.1 Cadriciel MPU pour espaces mémoire imbriqués sécurisés et flexibles

A.1.1 Flexibilié pauvre dans les solutions courantes d'isolation basées sur MPU

Les solutions existantes de l'état de l'art ne proposent qu'une isolation plane (au mieux deux niveaux) et la MPU n'est pas librement configurable. Nous différencions ces systèmes existants selon leur dynamisme et leur flexibilité dans le Tableau A.4. Le dynamisme des solutions dépend de la reconfiguration de la MPU lors de l'exécution.

[1] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/~/tree/addMemoryBlock_proof/

OS/noyau/ hyperviseur/outil	***Dynamisme***			***Flexibilité***	
	Reconfiguration MPU	Modèle de permission	Extension mémoire	Nature de la compartimentation	Imbrication
TrustLite [111]	✗	immuable	✗	processus	✗
JC VM endurcie [30]	✓	immuable	✗	processus	✗
EwoK [25]	✓	immuable	✗	processus	✗
µVisor	✓	immuable	✗	*thread*	✗
ACES [32]	✓	immuable	✗	générique	✗
OPEC [195]	✓	immuable	✗	générique	✗
TockOS [175]	✓	immuable	(✓)	processus	✗
Mbed [20]	✓	(variable)	(✓)	*thread*	✗
RIOT (MPU) [22]	✓	(variable)	(✓)	*thread*	✗
Zephyr (MPU) [189]	✓	(variable)	(✓)	*thread*	✗
FreeRTOS-MPU [74]	✓	(variable)	(✓)	tâche	✗
Pip-MPU	✓	variable	✓	générique	✓

Table A.4: Comparaison de la compartimentation dans les systèmes basés sur MPU. Pip-MPU est introduit au Chapitre 7 comme une spécialisation du cadriciel présenté dans le Chapitre 6.

Certains systèmes se contentent d'une configuration globale statique, comme TrustLite [111]. Tous les domaines protégés se partagent alors cette même configuration. Une configuration statique est un inconvénient dans l'environnement dynamique des objets connectés qui limite fortement des mise-à-jour de permissions d'accès, de services ou de chargement dynamique d'applications requérant une disposition mémoire différente.

D'autres systèmes bénéficient au contraire d'une reconfiguration de la MPU, qui s'adapte aux besoins de protection courants. La MPU est principalement dédiée à la protection d'un composant logiciel à la fois, et reconfigurée lors d'un changement de contexte. Chaque domaine protégé dispose de l'entièreté de la MPU pour soi. Bien que la MPU soit reconfigurée lors de l'exécution, elle peut soit suivre un modèle de permission fixé au préalable et donc non modifiable (la plupart des systèmes de l'état de l'art tels que µVisor, MINION, MultiZone, ACES, OPEC, EwoK, Java Card VM endurcie, TockOS [11, 105, 141, 32, 195, 25, 87, 175]) ; ou bien librement modifier ce modèle de permission (tels que Mbed [20], RIOT (MPU) [22], FreeRTOS-MPU [74] et Zephyr (MPU) [189]), cependant la MPU est alors utilisée comme outil de protection mémoire et non pour mettre en place une isolation.

Pour la plupart de ces systèmes, les composants logiciels disposent d'un domaine protégé de taille fixe qui ne peut être étendu ensuite. Une exception est faite pour TockOS qui alloue dynamiquement, et de façon limitée, un peu de mémoire aux processus en libérant des sous-régions initialement désactivées. Bien entendu, avec un modèle de permission variable, il est possible d'étendre sa mémoire, même de façon non sécurisée, sans parler du fait qu'ils ont les privilèges de faire sauter complètement le modèle de permission.

Chaque solution utilise la MPU à sa manière rendant l'ensemble très hétéroclite, mais aussi très créatif. En revanche, certains systèmes sont fortement liés à leur plateforme matérielle, par exemple en dépendant des sous-régions de l'architecture ARMv7, qui disparaissent dans la version actuelle de l'architecture ARMv8. Par ailleurs, d'autres systèmes ont plutôt décidé de modifier le matériel, comme TrustLite [111], ce qui inclut un coût de conception supplémentaire et limite la portabilité de la solution.

Nous observons un manque de flexibilité dans les systèmes actuels. Le compromis

entre flexibilité, performances et sécurité oblige certains systèmes à revoir drastiquement leur solution de sécurité idéale et se font imposer de fortes limitations par la MPU. Ils sont de plus presque tous conçus pour l'architecture ARMv7 et non compatibles avec les versions récentes qui existent pourtant depuis plusieurs années. Par ailleurs, les solutions sont peu génériques et peu portables sur d'autres architectures.

Pour répondre à ce manque de flexibilité, nous proposons un cadriciel qui redonne une liberté quasi-complète aux développeurs de disposer de la MPU comme ils le souhaitent pour appliquer leur propre politique de sécurité, de façon locale, tout en étant restreint par une politique de sécurité globale.

A.1.2 Architecture de sécurité pour espaces mémoire imbriqués

Nous définissons ci-après notre architecture de sécurité pour mettre en place un schéma flexible d'imbrication d'espaces mémoire.

Contrôle ubiquitaire de la MPU

Le système flexible permet à n'importe quel composant logiciel de maîtriser son propre espace mémoire lors de son exécution par le contrôle de la MPU. En particulier, il peut définir ses propres régions mémoire accessibles et créer et gérer des **espaces mémoire imbriqués** (ou **sous-espaces**). De même, ces derniers héritent de cette capacité.

Configuration centralisée de la MPU par appels système

La MPU ne peut être configurée en mode privilégiée. Cependant, ce niveau de privilège est trop élevé pour les composants logiciels du système, qui comprend des composants qui ne sont pas de confiance. En effet, ils pourraient en profiter pour attaquer d'autres composants en s'octroyant l'accès à leur espace mémoire protégé.

Pour résoudre ces conflits, nous proposons de donner l'exclusivité du rôle de configuration de la MPU à une entité (privilégiée) unique de l'espace noyau. Tous les autres composants logiciels sont rejetés dans l'espace utilisateur, sans privilèges particuliers. Afin de leur permettre tout de même de contrôler la MPU, selon le point précédent, il leur faut passer par l'entité privilégiée par appels système. Cette entité privilégiée doit être de confiance au vue des risques encourus en cas de détournement d'usage ou de mauvaise configuration de la MPU.

Dans les solutions courantes, ce niveau de personnalisation de la MPU n'est jamais atteint et notre proposition ne suppose aucune région MPU réservée pour un usage particulier ou liée à des droits d'accès spécifiques.

A.1.3 Personnalisation de la politique de sécurité

La personnalisation de la MPU accordée aux composants logiciels de l'espace utilisateur leur permettent d'implémenter des modèles de permission d'accès locaux à leurs espaces mémoire imbriqués. Par exemple, cela peut être d'imposer le principe $W \hat{} X$, ou de la mémoire partagée, ou encore la protection personnelle de composants internes, ou bien des espaces mémoire temporaires.

Cependant, il faut que ces composants soient limités un minimum dans leur pouvoir d'action par l'entité privilégiée, sinon nous retomberions dans une situation où ils

auraient les pleins pouvoirs sur la MPU et pourraient en profiter pour attaquer les autres composants. Il doit donc y avoir une restriction globale définie au sein de l'entité privilégiée, implémentée par les services, que les permissions locales doivent respecter en tout temps. Par exemple, il pourrait s'agir d'une interdiction de modifier les droits d'accès initialement octroyés lors de la création d'un sous-espace mémoire.

A.1.4 Interface de Programmation Applicative (IPA)

Les composants logiciels interagissent avec l'entité privilégiée centralisée en charge de la configuration MPU par les appels système. Ceux-ci fournissent des services de gestion des sous-espaces mémoire. Ces services sont non préemptibles et rendent l'IPA à la fois **dynamique** et **flexible**.

IPA dynamique

S'agissant du dynamisme, il convient de laisser un espace mémoire se transformer en cours d'exécution selon la volonté de l'utilisateur. Celle-ci peut partager sa propre mémoire à des sous-composants pour étendre la mémoire d'un sous-espace, symétriquement récupérer cette mémoire, afin de redonner de la mémoire à un espace mémoire épuisé, ou transmettre des messages, ou encore donner l'accès à des périphériques. Elle peut également modifier les caractéristiques de bloc mémoire, dont leur taille par l'introduction d'une fonctionnalité de découpage mémoire, et leurs permissions d'accès qui ne peuvent dépasser les permissions originelles.

L'état courant d'un espace mémoire est stocké dans des **structures de métadonnées**. Les métadonnées enregistrent chaque modification de l'espace mémoire. Pour des raisons de sécurité, ces métadonnées ne sont pas accessibles à l'utilisateur qui pourrait sinon détourner la politique de sécurité globale en les modifiant convenablement. L'IPA dispose de sept appels système pour le dynamisme :

- `add, remove`: ces appels système étendent, respectivement restreignent, l'espace mémoire d'un sous-espace imbriqué. Ils peuvent être utilisés pour déplacer des bouts de mémoire et notamment isoler du code ou transmettre des messages.

- `prepare, collect`: ces appels sytème de gestion d'espace mémoire mettent en place des structures de métadonnées, respectivement les retirent. Ils doient être appelés avant d'ajouter, ou respectivement enlever, de la mémoire, afin de séparer les actes d'extension mémoire des opérations de gestion qui les rendent possibles. Leur contenu n'est pas accessible à l'utilisateur.

- `cut, merge`: les blocs mémoire vont grandir et se réduire selon les besoins de l'application. Le découpage et la fusion de blocs permettent de redéfinir des blocs mémoire attribués à un composant logiciel, dans l'optique de ne pas partager l'ensemble d'un bloc par exemple.

- `activate`: cet appel système sélectionne un bloc mémoire et le configure dans la MPU. L'utilisateur choisit quelle région MPU doit abriter le bloc.

IPA flexible

L'utilisateur peut former autant de sous-espaces que souhaités, sans avoir besoin de fixer la structure globale lors de la compilation ou du démarrage. L'IPA propose deux appels système qui rendent possibles la flexibilité par la création d'espace mémoire imbriqués :

- `create, delete`: ces appels système créent, respectivement détruisent, un espace mémoire imbriqué. La création consiste à déclarer le nouveau sous-espace dans un bloc de métadonnées, tandis que la destruction redonne à l'espace mémoire parent tous les blocs qu'il avait préalablement partagés. Le bloc contenant la structure de métadonnées créé à l'occasion d'un `create` n'est plus accessible à l'utilisateur.

A.1.5 Structures de données et virtualisation MPU

Nous détaillons maintenant les structures de métadonnées nécessaires au fonctionnement de la compartimentation flexible.

Structure de MPU *virtuelle* (ou blocs) Les blocs mémoire d'un espace mémoire ne sont pas tous configurés dans la MPU. En effet, ils peuvent être plus nombreux, surtout par la fonctionnalité de découpage, que le nombre de régions MPU. L'utilisateur sélectionne les blocs à activer dans la MPU. En fait, la liste des blocs d'un espace mémoire est séparée de la liste des blocs configurés dans la MPU. C'est comme si la MPU était *virtualisée*.

La liste de l'ensemble des blocs mémoire est contenue dans la structure de métadonnées MPU *virtuelle*, où sont enregistrées leurs caractéristiques : adresses de début et de fin, permissions d'accès, accessibilité (éligible à être configuré dans la *vraie* MPU) et d'existence (n'est pas un emplacement vide). Tous les emplacements (slots) vides sont chainés entre eux. Une structure de MPU *virtuelle* accueille un nombre limité de blocs, et alors il est possible d'étendre la liste en ajoutant une structure chainée aux autres par le service `prepare`, ou au contraire la réduire par `collect`.

Structure de partage Pour chacun des blocs de la structure de MPU *virtuelle* est associé une entrée de partage, sauvegardée dans une liste séparée. Cette entrée indique si le bloc associé est partagé avec un sous-espace, duquel il s'agit et la localisation du bloc s'il est partagé. De même, `prepare` et `collect` gère la taille de la liste.

Structure de découpe Chaque bloc découpé est mis en relation avec son bloc originel pour tracer les découpages. De même, cette structure est gérée par `prepare` et `collect`.

Structure principale La structure principale stocke les liens vers toutes les autres structures d'un espace mémoire donné, ainsi qu'un lien vers l'espace mémoire parent s'il en a un, la liste des blocs configurés dans la *vraie* MPU, le nombre d'emplacements vides et un pointeur vers le premier emplacement vide. Cette structure est créée, respectivement détruite et par conséquence tous les sous-espaces mémoire imbriqués que l'espace mémoire considéré abrite, par les appels système `create` et `delete`.

A.2 Pip-MPU, adaptation du noyau Pip par spécialisation du cadriciel

Jusqu'à ce point, nous nous sommes concentrés sur le module flexible et sécurisé de gestion MPU. Cependant, ce module ne suffit pas pour faire tourner des vrais programmes. Nous démontrons alors une utilisation possible du cadriciel avec le proto-noyau Pip que nous adaptons aux systèmes basés sur MPU puisqu'il repose actuellement sur la MMU. Nous appelons cette adaptation **Pip-MPU**. Pip a besoin d'une forte flexibilité et possède une politique de sécurité globale très restrictive.

A.2.1 Pip

Pip est un noyau spécialisé en gestion mémoire et en changement de contexte. Il est seul privilégié dans le système, tous les autres composants logiciels tournent dans l'espace utilisateur. Pip démontre de fortes garanties de sécurité avec une propriété d'isolation formellement vérifiée avec appui de l'assistant de preuves Coq [95].

Pip base son modèle de sécurité sur un arbre de partitionnement. Une **partition** est typiquement un processus, une tâche ou n'importe quelle unité exécutable avec ses données. L'espace mémoire d'une partition est composée de pages mémoire. Une partition peut créer une ou plusieurs sous-partitions, ce qui forme un arbre généalogique. Toutes les partitions descendent de la **partition racine** et sont dynamiquement créées durant le cycle de vie du système.

Les partitions sont reliées entre elles par des relations **parent-enfant**. Une **partition parente** peut avoir des **descendants**, de même qu'une **partition enfant**. Pour un descendant, la partition parente est un **ancêtre**. Les partitions qui ont le même parent sont des **partitions soeurs**.

Le modèle de sécurité de Pip garantit trois propriétés de sécurité, illustrées dans la Figure A.2:

- **Partage Vertical** : une partition parente peut partager des bouts de sa propre mémoire à l'une de ses partitions enfant.
- **Isolation horizontale** : une page mémoire ne peut être partagée par un parent qu'avec un unique enfant. Cette propriété garantit l'isolation mémoire stricte entre des partitions soeurs.
- **Isolation du noyau** : la mémoire attribuée à Pip, ainsi que les métadonnées qui stockent les attributs des espaces mémoire des partitions, ne sont pas accessibles aux partitions. Cette propriété protège le noyau et garantit qu'aucune partition ne peut changer sa configuration en accédant aux métadonnées.

A.2.2 Exigences pour Pip-MPU

Pip-MPU propose plus de services que l'entité privilégiée du cadriciel. Nous catégorisons en quatre catégories les exigences que doit suivre Pip-MPU pour valider l'adaptation : sécurité, performances, fonctionnelles et matérielles. Certaines exigences sont directement héritées de Pip tandis que d'autres ne sont seulement requises pour l'adaptation à des appareils contraints en ressources.

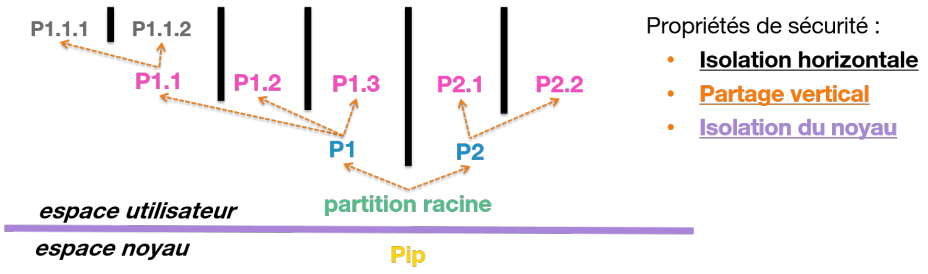


Figure A.2: Modèle de sécurité de Pip.

Exigences héritées de Pip :

- **SecReq1: les propriétés de sécurité de Pip** Les propriétés de sécurité de Pip présentées précédemment doivent être garanties.
- **SecReq2:Protection mémoire matérielle** Les accés mémoire illégaux doivent être bloqués et identifiés par les composants matériels de protection mémoire. Seul l'espace noyau a suffisamment de privilèges pour les configurer.
- **SecReq3: Taille logicielle minimale** Le code de Pip doit être minimal afin de pouvoir être vérifié formellement, pour réduire la probabilité de vulnérabilités et pour faciliter la maintenance de la base de code.
- **SecReq4: Modifications des permissions d'accès limitées** Pip doit garantir que seule une partition parente peut gérer les permissions d'accès des blocs mémoire (en lecture, écriture, exécution). Pip doit garantir qu'une partition ne puisse pas élever ses droits toute seule sur elle-même ou ses partitions enfant.
- **FuncReq1: Partitions flexibles** L'arbre de partitionnement doit être librement déterminé lors de l'exécution. Toutes les partitions peuvent créer et isoler des sous-espaces issus de leur propre espace mémoire.
- **PerfReq1: Dégradation des performances raisonnables** Pip doit maintenir les exigences de performances existantes avant un portage sur Pip afin de convenir au monde concret. Cela inclut une phase de démarrage rapide qui ne doit pas impacter significativement la routine de démarrage.

Exigences spécifiques à Pip-MPU :

- **HWReq1: Protection mémoire par la MPU** Pip-MPU doit spécifiquement utiliser la MPU comme matériel de protection mémoire. Puisque la MPU n'est que disponible dans les appareils à fortes contraintes de ressources, le corollaire est que Pip-MPU ne ciblent que cette classe d'appareils..
- **HWReq2: Pas de modifications matérielles** Pip-MPU doit utiliser les composants matériels déjà commercialisés, sans les modifier. Ceci pour faciliter l'adoption et réduire le temps de développement.

Appel système	**Complexité**
create/delete	$O(s*k)$
cut/merge	$O(s*k)$
add/remove	$O(s)$
prepare/collect	$O(s*k)$
activate	$O(1)$

Table A.5: Analyse des complexités des appels système du cadriciel implémentés pour Pip-MPU. s représente le nombre structures de métadonnées d'une partition donnée (<8) et k est le nombre d'ancêtres de cette partition (attendu < 4).

- **PerfReq2: Temps d'exécution limité** L'analyse de complexité des algorithmes de Pip-MPU et l'implémentation du code doivent être compatibles avec des contraintes temps-réel pour convenir à de nombreux scénarios d'appareils contraints.
- **PerfReq3: Faible consommation mémoire** Pip-MPU doit laisser assez d'espace mémoire aux applications.
- **PerfReq4: Faible consommation énergétique** La consommation énergétique supplémentaire due à Pip-MPU doit rester raisonnable afin de ne pas pénaliser les appareils contraints alimentés par batterie.

A.2.3 Implémentation de Pip-MPU

Il y a une proximité évidente entre les structures de métadonnées du cadriciel et celles de Pip qui nous permettent de valider un certain nombre d'exigences.

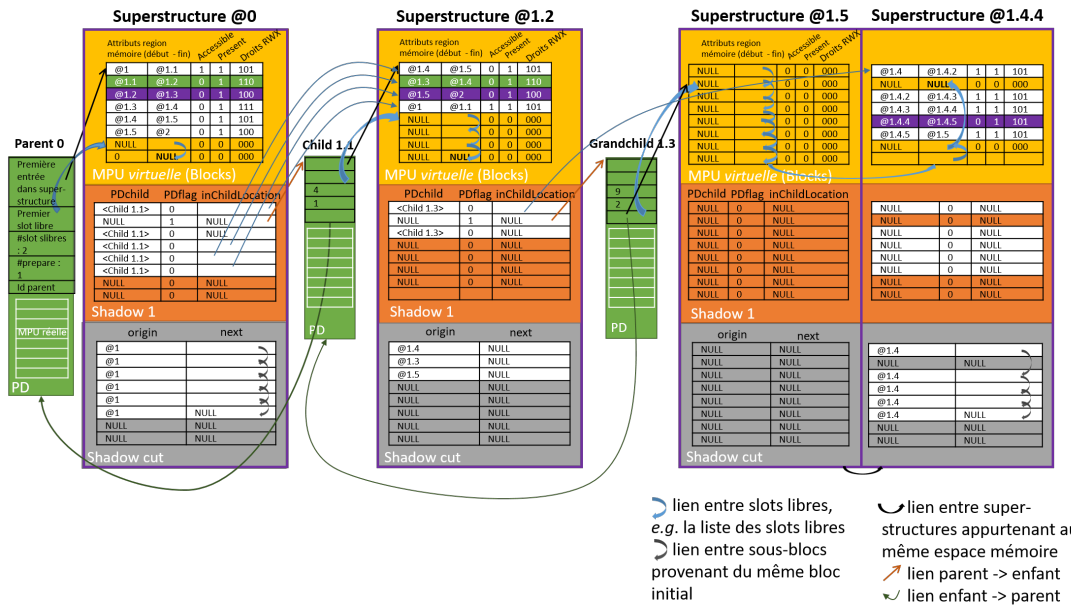
De plus, nous spécialisons le cadriciel pour satisfaire les propriétés de sécurité de Pip. Tout d'abord, nous restreignons le partage de blocs mémoire avec un sous-espace en ajoutant un champ unique identifiant le partage. Nous obligeons également dans l'implémentation du service de partage que les permissions d'accès ne soient jamais élevées. De plus, nous adoptons la même nomenclature que Pip et fusionnons les structures de MPU *virtuelle* (devient **Blocks**), de partage (devient **Shadow 1**) et de découpe (devient **Shadow Cut**), tandis que la structure principale est laissée à part (et devient **Partition Descriptor**). Les relations parent-enfant sont illustrées sur deux niveaux dans la Figure A.3.

Le flux de travail et l'organisation du code reprend les codes de Pip et les services spécialisés du cadriciel sont implémentés directement dans l'assistant de preuves Coq.

Par ailleurs, le nombre de blocs mémoire au sein d'une partition a été limité à 64 par partition, ce qui mène à l'analyse de complexité de chaque appels système du cadriciel reportée dans le Tableau A.5.

A.2.4 Evaluation de Pip-MPU

Nous évaluons notre implémentation de Pip-MPU en exécutant une application au sein de notre prototype de Pip-MPU sur un microcontrôleur basé sur un processeur ARMv7 Cortex-M et en comparant l'exécution à une implémentation de référence sans Pip. L'objectif de cette évaluation est double : 1) Est-ce que la solution est utilisable



)

Figure A.3: Exemple de relations parent-enfant sur deux niveaux du point de vue des structures de métadonnées. Partition 0 (parent) est le parent de partition 1.3 (enfant de parent 0) qui est à son tour parent de partition 1.3 (grand-enfant de parent 0). Les blocs verts contiennent les structures PD de leurs enfants. Les blocs violets contiennent les super-structures de leurs enfants ou pour eux-mêmes (cas du petit-enfant 1.3). L'illustration montre que le bloc 1.4 a été découpé plusieurs fois (en 1.4.2, 1.4.3, 1.4.4 and 1.4.5) ce qui implique que le bloc original 1.4 est inaccessible dans les ancêtres. Pour chaque bloc mémoire des partitions, on observe aucune élévation des permissions d'accès par rapport à leur bloc originel, mais ne peut qu'être descendu (comme pour le bloc 1.3). Le petit-enfant 1.3 montre un cas d'école après un `prepare`: une nouvelle super-structure est initialisée en consommant le bloc 1.4.4 de la même partition 1.3, et la nouvelle super-structure est placée en tête de la liste chaînée des super-structures de la partition, le premier slot libre est remplacé par la première entrée de la nouvelle super-structure et toutes les entrées ont les valeurs par défaut exceptées la dernière entrée qui est reliée à l'ancienne liste des slots libres de la partition. Le bloc 1 est de la mémoire partagée entre le parent 0 et son enfant 1.1. Dans notre implémentation, nous considérons une structure de MPU *virtuelle* avec huit entrées. A noter, l'absence d'ordre dans la liste des slots libres du petit-enfant 1.3 du aux précédents `cut` et `merge`. Notons également la référence unique entre les super-structures 1.5 et 1.4.4 comparées aux multiples références dans le cadriciel entre chaque structure interne.

Empreinte mémoire en Flash (octets/SLOC de C)	
Taille totale de Pip-MPU	9544 / 4186
Empreinte mémoire en RAM (octets)	
Pile de Pip-MPU	516
Structures de métadonnées:	
- par partition	$640 + (\mathrm{B}\ mod\ 8) \times 512$
- minimum par partition	1152
- maximum par partition	4736
Déploiement (#cycles)	
Initialisation de Pip-MPU	Partition racine : 99022 / Partition enfant : 165582

Table A.6: Surcoût de Pip-MPU. Pour mesurer la taille de Pip-MPU et son empreinte mémoire, nous avons utilisé l'option de compilation -0s.

sur des objets contraints ? 2) Quels sont les surcoûts liés aux performances (cycles, consommation énergétique) et à la présence de la couche logicielle supplémentaire (taille, lignes de code, temps d'initialisation) pour quels gains de sécurité (mémoire accessible, opérations privilégiées) ?

Notre prototype tourne sur un kit de développement nRF52840 Nordic Semiconductor (nRF52840-DK)[131]. La puce embarque un processeur ARM Cortex-M4 (architecture ARMv7-M) s'exécutant à 64 MHZ et comprenant une mémoire flash de 1 Mo et une mémoire vive de 256 ko, ainsi que d'une MPU composée de 8 régions MPU. Nous conduisons des analyses statiques et dynamiques sur 4 applications issues de la suite de tests d'évaluations Embench IoT benchmark suite [72]: aha-mont64, crc32, nsichneu, primecount.

L'évaluation consiste en deux scénarios Pip, faisant tourner une application au sein 1) de la partition racine ou 2) d'une partition enfant. De façon identique à Pip, la partition racine est lancée par Pip-MPU à la fin de son initialisation et est l'unique partition au démarrage. La partition racine initialise et lance ensuite elle-même des partitions enfant. Ces partitions enfant peuvent également créer des partitions enfant, et ainsi de suite. Toutes les partitions s'exécutent dans l'espace utilisateur. Nous avons comparé chaque scénario à notre scénario de référence qui est l'exécution de la même application en mode privilégié, sans Pip et lancée après la même phase d'initialisation. L'application est régulièrement interrompue par une interruption SysTick toutes les 10 ms. L'interruption déclenche une routine qui rend la main immédiatement dans le cas de notre scénario de référence, ou alors traverse la routine d'interruption de Pip-MPU dans les scénarios Pip. Une expérience associe une application avec un scénario.

Nous avons écrit des scripts Python pour piloter la phase d'évaluation, inspirés par et adaptés selon les scripts et les outils proposés par Embench and BenchIoT [2]. Les résultats finaux présentent le surcoût de Pip-MPU dans le Tableau A.6, et le surcoût induit sur les performances de l'application et les gains de sécurité obtenus dans le Tableau A.7.

Les lignes de code source (*SLOC*, *Source Lines of Code*) sont le nombre de lignes de code source C après suppression de tous les commentaires et de toutes les lignes

vides de tous les fichiers source C par l'utilisation de l'option GCC `-fpreprocessed`. Les lignes comptabilisées incluent celles qui ne contiennent que des accolades, des variables globales ou encore les paramètres de fonction pouvant s'étaler sur plusieurs lignes (même si peu fréquent). Le Tableau A.6 présente les SLOC et la taille (en octets) de Pip-MPU, qui comporte ses services principaux et ses routines d'interruption, ainsi que les fonctions bas-niveau d'accès au matériel.

L'utilisation de la pile est mesurée sur les pile principale (*main stack*) et utilisateur (*user stack*) en les marquant au préalable par une valeur prédéfinie. A la fin du test, la dernière position comprenant la valeur prédéfinie témoigne de l'usage. En plus des métadonnées requises pour la partition racine, l'empreinte mémoire de Pip-MPU comprend les métadonnées utiles pour la création d'autres partitions, dont la partition enfant de notre scénario. Les métadonnées des partitions comprennent les structures encapsulant la liste des blocs mémoire et leurs attributs, ainsi que des données globales à la partition. Lorsque le nombre de blocs d'une partition augmente, ces blocs sont référencés dans des structures de taille S. Chaque structure peut référencer un nombre constant de blocs C. Ainsi, pour une partition de B blocs, nous obtenons l'empreinte mémoire de ses métadonnées par la formule

$$K + (B \bmod C) \times S \; octets$$

avec K une taille de métadonnées incompressible. Dans notre implémentation, $K = 640$, $C = 8$, $S = 512$. Puisque chaque partition a besoin d'une structure de métadonnées au minimum pour référencer les premiers blocs, l'empreinte mémoire d'une partition en RAM sera d'au minimum 1152 octets (liste initiale de $B = 8$ blocs). Par ailleurs, le nombre de structures de métadonnées pour une partition donnée est limité à $MaxS$ lors de la compilation, ce qui implique une empreinte mémoire maximum de $K + MaxS \times S$ pour cette partition. Pour notre système, $MaxS = 8$ par partition, soit $640+8\times512 = 4736$ octets. De plus, $MaxS$ dicte le nombre de blocs qu'une partition peut référencer au maximum par $C \times MaxS$, soit ici 64 blocs.

Pour obtenir les métriques de performances du Tableau A.7, nous exécutons l'application pour chaque scénario (référence, partition racine et partition enfant). Nous lançons le programme principal 3 fois consécutivement pour la même expérience, ce qui nous permet de récolter des données sur au moins 20 secondes (chaque application s'exécutant en 5-7 secondes). Nous lançons l'expérience 5 fois et nous effectuons ensuite des statistiques (moyenne μ et écart-type σ) que nous moyennons pour chaque application. Le surcoût indiqué est la moyenne du surcoût observé pour chaque scénario comparé au scénario de référence.

Le nombre de cycles est obtenu par l'unité du processeur *Data Watchpoint and Trace* (DWT). Nous initialisons le compteur juste avant le lancement de l'application et nous récupérons le nombre de cycles écoulés après la fin de la phase d'initialisation ainsi que lorsque l'application a terminé son travail. La fin de la phase d'initialisation marque le début de la phase de test qui lance l'application. De plus, le scénario de référence s'exécute toujours en mode privilégié, ce qui correspond à 100% de cycles privilégiés. A l'inverse, dans les scénarios Pip (application s'exécutant dans la partition racine ou une partition enfant), les cycles privilégiés sont tous ceux mesurés dans les routines de Pip-MPU. Nous indiquons le ratio du nombre de cycles privilégiés sur le nombre total

Métriques	Partition racine	Partition enfant
Cycles		
Cycles :		
i) au total	$\mu = 76302131$	$\mu = 74538344$
	$\sigma = 67494444$	$\sigma = 73634323$
	(+16.31%)	(+16.4%)
ii) pendant le test	$\mu = 76203107$	$\mu = 74372762$
	$\sigma = 67495112$	$\sigma = 73634647$
	(+16.29%)	(+16.36%)
Ratio des cycles privilégiés sur le total des cycles :		
i) au total	$\mu = 0.86\%$	$\mu = 0.92\%$
	$\sigma = 3.8 \times 10^{-5}\%$	$\sigma = 3.3 \times 10^{-5}\%$
	(-99.14%)	(-99.08%)
ii) pendant le test	$\mu = 0.87\%$	$\mu = 0.92\%$
	$\sigma = 3.9 \times 10^{-5}\%$	$\sigma = 3.3 \times 10^{-5}\%$
	(-99.13%)	(-99.08%)
Consommation énergétique pendant la phase de test		
Totale	$\mu = 24.76\ mJ$	$\mu = 26.6\ mJ$
	$\sigma = 22.42\ mJ$	$\sigma = 23.00\ mJ$
	(+16.7%)	(+18.4%)
dont causée par la MPU	$\mu = 0.05\ mJ$	$\mu = 0.07\ mJ$
	$\sigma = 0.16\ mJ$	$\sigma = 0.11\ mJ$
	(+0.03%)	(+0.04%)
Securité		
Ratio de la mémoire accessible depuis l'application sur la mémoire totale:		
- Flash (code)	99.0%	6.27%
	(-1.0%)	(-93.73%)
- RAM (données)	99.35%	1.9%
	(-0.65%)	(-98.1%)

Table A.7: Comparaison des performances et de la sécurité par rapport au scénario de référence. L'application est soit exécutée au sein de la partition racine, soit dans la partition enfant.

de cycles 1) depuis le démarrage de la partition racine 2) seulement durant l'exécution de l'application.

Les zones de mémoire accessible représentent la mémoire que la partition peut accéder. L'application dans le scénario de référence a accès à toute la mémoire, soit 100%, tandis que dans les scénarios Pip les zones accessibles sont restreintes aux blocs de l'espace mémoire. Pour la partition racine, la mémoire accessible correspond à toute la mémoire, en retirant Pip-MPU et les composants de démarrage. A partir de là, la partition racine, ainsi que n'importe quelle autre partition parent, décide quels blocs mémoire sont partagés avec les partitions enfant, contrôlant de cette manière leurs zones de mémoire accessible.

La consommation énergétique est mesurée en utilisant un Power Profiler Kit I (PPKI) [132] monté sur la puce nRF52840-DK. Le PPK produit des mesures de courant à 77 kHz en moyennant sur 4 mesures, que nous multiplions par une tension fixe et que nous intégrons sur le temps écoulé afin d'obtenir la consommation énergétique totale. Etant donné l'utilisation du *semihosting* pour renvoyer les données de performance à l'ordinateur pour analyse, le débuggeur reste actif. Néanmoins, aucune entrée/sortie n'a lieu pendant la phase de test.

Pip-MPU prend respectivement 1664 octets (données, pile, structures de métadonnées de la partition racine) et 9544 octets (code) des 256 Ko de mémoire vive (3.3%) et du 1 Mo de mémoire flash (3.8%). Ainsi Pip-MPU laisse aisément de la place pour des applications complexes (exigence *PerfReq3*). Cela démontre également sa minimalité (exigence *SecReq3*. La métrique de mémoire accessible montre l'étendue de la surface d'attaque. Pip-MPU réduit cette surface d'attaque de 99% pour la partition racine. Le pourcent restant correspond à la mémoire réservée pour Pip-MPU, donc isolée et protégée. L'empreinte mémoire due à la création d'une partition enfant s'élève à 1Ko dans notre implémentation. Cependant, ce minimum ne comprend pas les entrées réquisitionnées dans la partition parente pour stocker les structures de métadonnées de l'enfant. Si ces entrées ne sont pas suffisantes, il faudrait alors créer de nouvelles structures de métadonnées ce qui consommerait 512 octets supplémentaires. Le nombre d'entrées des structures est défini par le développeur lors de la compilation et pourrait donc être ajusté pour correspondre au mieux au cas d'usage.

De plus, nous observons une dégradation des performances de 16% causée majoritairement par la séquence de restauration du contexte lors de la réception d'une interruption SysTick. Cette valeur n'a de sens que pour notre scénario de test et est attendue plus important dans le cas d'un OS porté sur Pip à cause de multiples interruptions. Nous avons identifié certaines optimisations qui permettraient de réduire cette dégradation si elle s'avère trop importante.

Le surplus de consommation énergétique est de 17-18%. La MPU n'explique que 0.02-0.2% de ce total en fonction des scénarios. Cela indique que la MPU impacte peu la consommation ce qui était pourtant un argument avancé pour soutenir la non-utilisation de celle-ci de façon globale dans les systèmes contraints [194].

Ainsi, notre analyse préliminaire et l'évaluation de Pip-MPU ont démontré une parfaite concordance avec les exigences attendues. De plus, notre prototype n'est pour l'instant compatible que pour des architectures ARMv7. Cependant, la MPU est présente dans la majorité des appareils basés sur ARM Cortex-M3/4/7 [33] et de plus, nous revendiquons une compatibilité étendue à ARMv8 ainsi que pour des composants

équivalents d'autres architectures tel que RISC-V Physical Memory Protection (PMP) [176].

Par ailleurs, Pip-MPU protège des attaques identifiées dans notre modèle d'attaque. La MPU protège le système dès son démarrage et avant le lancement de la partition racine. La compartimentation flexible mise en place par Pip-MPU empêche un attaquant qui aurait réussi à compromettre un composant de se propager dans l'appareil en-dehors du composant compromis.

Pour cela, nous faisons confiance au chargement de Pip-MPU et au lancement correct de la partition racine, ainsi que la chaine de compilation, les IDEs et outils utilisés et la plateforme matérielle disposant d'une MPU fonctionnant tel que compris par les spécifications. La plupart des systèmes reposent sur cette même base de confiance, il est donc réaliste.

A.3 Vérification formelle de la propriété d'isolation de Pip-MPU

Malgré l'argumentation précédente soutenant que l'on pouvait faire confiance à Pip-MPU, cela peut ne pas être suffisant dans le cas de systèmes critiques ayant besoin de fortes garanties.

Les méthodes formelles apportent ce degré de confiance par des démonstrations mathématiques de la sécurité vérifiées par ordinateur, comme cela a été fait pour Pip. Pip utilise la logique de Hoare [90] pour conduire sa preuve pour vérifier que ses propriétés de sécurité (*cf.* paragraphe A.2.1: Partage Vertical, Isolation Horizontale et Isolation du noyau) sont des **invariants** (propriétés satisfaites à la fin de l'exécution du service si l'exécution est terminée et que les propriétés étaient vraies au début de l'exécution). Les preuves sont réalisées au sein de l'assistant de preuve Coq [95].

A.3.1 Objectifs de preuve dans Pip-MPU

Pip-MPU tend à vérifier l'invariant d'isolation sur ses services de la même manière que Pip. Les **propriétés de sécurité** (décrites plus haut : Partage Vertical, Isolation Horizontale et Isolation du noyau) reposent sur des propriétés structurelles des structures de métadonnées, appelées **propriétés de cohérence**. En effet, les propriétés de sécurité sont définies selon l'arbre de partitionnement lui-même sous-jacent aux structures de métadonnées. Il convient alors de vérifier deux groupes d'invariants : les propriétés de sécurité et les propriétés de cohérence.

Definition A.3.1 (Invariant d'isolation)**.** L'invariant d'isolation de Pip-MPU est la conjonction de ses propriétés de sécurité S et de ses propriétés de cohérence C, soit $S \bigwedge C$.

Ainsi, il faut pour chaque service prouver le triplet de Hoare suivant :

$\{S\&C\}$
service Pip-MPU (*paramètres...*)
$\{S\&C\}$

Concernant les propriétés de cohérence C, ils sont du nombre de vingt-sept, classées en trois groupes : les propriétés sur les entrées mémoire, les propriétés manipulant des structures abstraites telles des listes et tableaux, et les propriétés qui régissent les relations entre partitions. Toutes les propriétés sont dépendantes de l'état courant, modélisé comme l'association de la partition courante et la mémoire.

Propriétés sur les entrées mémoire :

- nullAddrExists s
- wellFormedFstShadowIfBlockEntry s
- wellFormedShadowCutIfBlockEntry s
- PDTIfPDFlag s
- AccessibleNoPDFlag s
- FirstFreeSlotPointerIsBEAndFreeSlot s
- BlocksRangeFromKernelStartIsBE s
- KernelStructureStartFromBlockEntryAddrIsKS s
- sh1InChildLocationIsBE s
- StructurePointerIsKS s
- NextKSIsKS s
- NextKSOffsetIsPADDR s

Propriétés manipulant des ensembles abstraits :

- NoDupInFreeSlotsList s
- freeSlotsListIsFreeSlot s
- DisjointFreeSlotsLists s
- inclFreeSlotsBlockEntries s
- DisjointKSEntries s
- noDupKSEntriesList s
- noDupMappedBlocksList s
- noDupUsedPaddrList s
- MPUInAccessibleBlocks s

Propriétés sur les relations inter-partitions :

- currentPartitionInPartitionsList s
- noDupPartitionTree s
- isParent s
- isChild s
- accessibleChildPaddrIsAccessibleIntoParent s
- sharedBlockPointsToChild s

A.3.2 Résultats

Il y a deux types de services : ceux qui ne modifient pas l'état courant (uniquement des instructions de lecture) et ceux qui le modifient (instructions d'écriture).

Pour le premier type de services, la preuve est triviale. En effet, puisque l'état n'est pas modifié, il est le même à la fin de l'exécution du service qu'au début où les propriétés étaient satisfaites. Nous avons vérifié de cette manière que la propriété d'isolation était satisfaite dans les services `findBlock` et `readMPU`.

Pour le second type de services, la preuve est difficile. En effet, en modifiant l'état, il faut également s'assurer que les propriétés sont toutes satisfaites dans l'état modifié. Nous avons montré informellement, puis formellement dans l'assistant de preuve Coq[2] que les propriétés d'isolation étaient satisfaites dans le service `addMemoryBlock`.

Nos preuves sont développées en utilisant CoqIDE 8.13.1 s'exécutant sur la distribution ArchLinux 5.18.1. La plateforme matérielle pour les preuves a été un ordinateur HP ZenBook G4 17 possédant un processeur 64-bit Intel i7 7820HQ 4 coeurs, 16 Go de RAM, 16 Go de swap, et 155 Go d'espace disque utilisable.

Le Tableau A.8 relate le statut courant du développement des preuves et les résultats obtenus ainsi que certaines métriques de preuve courantes. Ce tableau présente tout d'abord le nombre de services prouvés et les métriques sur l'invariant d'isolation. Il relate également le temps passé pour la formalisation et pour le développement des preuves des trois services prouvés. Il présente aussi le ratio de lignes de code sur lignes de preuve ou tactiques Coq, ainsi que l'estimation en personne-temps de chaque service et le ratio par lignes de code vérifiées.

A.3.3 Hypothèses

La validité des preuves dépendent d'un certain nombre d'hypothèses, dont les outils utilisés (Coq, digger, IDEs), la validité du modèle matériel, la correspondance entre les primitives bas-niveau sur lesquels reposent l'écriture des services dans Coq et leur équivalent C réellement compilé, l'implémentation matérielle et la correction de sa spécification, l'expression juste de l'invariant d'isolation et la phase de démarrage qui satisfait la pré-condition des triplets de Hoare de chaque service.

A.3.4 Découverte de bugs

La vérification formelle de `addMemoryBlock` a révélé un bug. En effet, la phase de vérification des paramètres omettait de vérifier qu'un bloc n'était pas déjà partagé avant de le faire, ce qui amenait à une situation où des partitions soeurs pouvaient disposer du même bloc mémoire, ce qui est interdit par la propriété de sécurité de l'isolation horizontale. Le bug a été découvert par l'impossibilité de conclure la preuve de cette propriété. La découverte de bugs est fréquente dans les projets de vérification formelle [99, 29, 109].

[2] https://gitlab.univ-lille.fr/2xs/pip/pipcore-mpu/-/blob/addMemoryBlock_proof/proof/invariants/AddMemoryBlockSecProps.v

Statut de la preuve				
Services				
# services	15			
# services entièrement vérifiés	2			
Invariants				
# propriétés de cohérence	27			
# propriétés de sécurité	3			
Total propriétés	29			
SLOC propriétés de cohérence	129			
SLOC propriétés de sécurité	14			
Total SLOC propriétés	143			
Développement des preuves				
Formalisation				
Temps de compréhension de la spécification matérielle	2 mois			
Durée de conception	9 mois (dont 2 mois de simulation en Python)			
Temps d'implémentation du modèle Coq	2 mois			
Durée d'implémentation des services en Coq depuis pseudo-code	2 mois			
	readMPU	**findBlock**	**addMemoryBlock**	
# SLOC (sans HAL)	11	14	25	
Preuves	*Toutes les propriétés*		*Propriétés de cohérence*	*Propriétés de sécurité*
# LdP (Lignes de Preuve)	76	81	6143	1516
# tactiques	90	93	6838	759
Ratio #LdP/SLOC	6.9	5.8	246	60.4
Ratio #tactiques/SLOC	8.18	6.6	273	30.4
Durée de développement des preuves	2 jours	4 heures	7 mois (en cours)	4 jours
personne-mois (estimation)	0.09	0.02	5.5	0.18
Ratio personne-mois/SLOC	0.008	0.0014	0.22	0.0072
Temps de compilation de la preuve	0.5s	0.4s	939s (15.65 min)	127s (2.1 min)
Empreinte mémoire pendant la compilation de la preuve	400 Mo	400 Mo	5000 Mo	900 Mo
Utilisation du CPU pendant la compilation de la preuve	100%	110%	115 %	105%
Couverture de preuve (estimation)	100%	100%	90%	100%

Table A.8: Efforts de preuve et statut.

A.4 Techniques de preuves et évolution du processus de vérification formelle

Pip-MPU a hérité du cadre formel de Pip. Cependant, toute la formalisation a dû être modifiée, ainsi que les propriétés de sécurité puisque Pip basait son raisonnement sur l'unité d'une page mémoire, qui n'existe pas dans Pip-MPU. Très tôt, la base de preuve a été fondamentalement et profondément changée.

A.4.1 Formalisation

Cependant, il a été souhaité de garder de fortes similitudes là où cela faisait sens, par exemple les propriétés de sécurité. Malgré une base formelle différente, elles sont quasiment identiques dans leur expression, avec des différences plus prononcées dans les définitions des fonctions internes.

```
(** PIP (MMU) **)
(** THE VERTICAL SHARING PROPERTY
↪ *)
Definition verticalSharing s : Prop
↪ :=
forall parent child : page ,
  In parent (getPartitions
  ↪ pageRootPartition s) ->
  In child (getChildren parent s) ->
  incl (getUsedPages child s)
  ↪ (getMappedPages parent s).

(** THE ISOLATION PROPERTY BETWEEN
↪ PARTITIONS *)
Definition partitionsIsolation  s :
↪ Prop :=
forall parent child1 child2 : page ,
  In parent (getPartitions
  ↪ pageRootPartition s)->
  In child1 (getChildren parent s) ->
  In child2 (getChildren parent s) ->
  child1 <> child2 ->
  disjoint (getUsedPages child1
  ↪ s)(getUsedPages child2 s).

(** THE KERNEL DATA ISOLATION
↪ PROPERTY *)
Definition kernelDataIsolation s :
↪ Prop :=
forall partition1 partition2,
  In partition1 (getPartitions
  ↪ pageRootPartition s) ->
```

```
(** PIP-MPU **)
(** THE VERTICAL SHARING PROPERTY
↪ *)
Definition verticalSharing s : Prop
↪ :=
forall parent child : paddr,
  In parent (getPartitions
  ↪ multiplexer s) ->
  In child (getChildren parent s) ->
  incl (getUsedPaddr child s)
  ↪ (getMappedPaddr parent s).

(** THE ISOLATION PROPERTY BETWEEN
↪ PARTITIONS *)
Definition partitionsIsolation  s :
↪ Prop :=
forall parent child1 child2 : paddr ,
  In parent (getPartitions
  ↪ multiplexer s) ->
  In child1 (getChildren parent s) ->
  In child2 (getChildren parent s) ->
  child1 <> child2 ->
  disjoint (getUsedPaddr child1 s)
  ↪ (getUsedPaddr child2 s).

(** THE KERNEL DATA ISOLATION
↪ PROPERTY *)
Definition kernelDataIsolation s :
↪ Prop :=
forall partition1 partition2 :
↪ paddr,
```

```
In partition2 (getPartitions
 ↪  pageRootPartition s) ->
disjoint (getAccessibleMappedPages
 ↪  partition1 s) (getConfigPages
 ↪  partition2 s).
```

```
In partition1 (getPartitions
 ↪  multiplexer s) ->
In partition2 (getPartitions
 ↪  multiplexer s) ->
disjoint (getAccessibleMappedPaddr
 ↪  partition1 s) (getConfigPaddr
 ↪  partition2 s).
```

En ce qui concerne les propriétés de cohérence, certaines sont différentes et attachées à leur contexte propre, ou bien se ressemblent fortement comme `isChild`, ou bien relatent de la même propriété mais, étant donné que Pip-MPU utilise une formalisation avec des types d'entrées mémoire plus riches, il allège son contexte de preuve, comme illustré avec dans `wellFormedFstShadow`.

```
(** PIP (MMU) **)
Definition wellFormedFstShadow (s :
 ↪  state) :=
forall partition,
In partition (getPartitions
 ↪  pageRootPartition s) ->
forall va pg pd sh1,
StateLib.getPd partition (memory s)
 ↪  = Some pd ->
StateLib.getFstShadow partition
 ↪  (memory s) = Some sh1 ->
getMappedPage pd s va= SomePage pg ->
exists vainparent,
 ↪  getVirtualAddressSh1 sh1 s va =
 ↪  Some vainparent.
```

```
(** PIP-MPU **)
Definition
 ↪  wellFormedFstShadowIfBlockEntry
 ↪  s :=
forall pa,
isBE pa s ->
isSHE (CPaddr (pa + sh1offset)) s.
```

```
(** PIP (MMU) **)
Definition isChild  s :=
forall partition parent : page,
In partition (getPartitions
 ↪  pageRootPartition s) ->
StateLib.getParent partition
 ↪  (memory s) = Some parent ->
In partition (getChildren parent s).
```

```
(** PIP-MPU **)
Definition isChild  s :=
forall partition parent : paddr,
In partition (getPartitions
 ↪  multiplexer s) ->
pdentryParent partition parent s ->
In partition (getChildren parent s).
```

A.4.2 Conduite de la preuve d'un service

Pip-MPU a appliqué sensiblement la même technique de preuve que Pip pour les services qui ne modifient pas l'état courant. Les instructions de lecture sont déroulées une par une et chaque instruction dispose de son lemme propageant l'invariant d'isolation jusqu'à la fin du service.

Cependant, la conduite de preuve change dans Pip-MPU pour les services modifiant l'état. Pip-MPU introduit le concept de ***checkpoints***, qui sont des instants dans la preuve où toutes les propriétés de cohérence sont vraies. Effectivement, certains autres instants, à cause de modifications de l'état, provoquent des états temporaires inconsistants où certaines propriétés de cohérence peuvent ne plus être vraies, jusqu'à la résolution finale aux checkpoints ou à la fin du service à prouver. Cela est différent dans Pip (MMU) qui suit un co-développement des services et de la preuve où ces états inconsistants seraient sans doute résolus par modification de l'ordre des instructions. Ce nouvel ordre impliquerait que toutes les propriétés de cohérence soient vraies à chaque instant. Pip-MPU au contraire s'abstient de modifier l'ordre par co-design pour deux raisons : la conception du système a été au préalable gelée pour commencer la phase de vérification formelle et parce que Pip-MPU s'appuie de façon prononcée et plus importante que Pip (MMU) sur la modularité de code. Cela est illustré par la fonction `insertNewEntry`, utilisée dans les services `addMemoryBlock` et `cutMemoryBlock`. Là où Pip (MMU) n'aurait pas pu généraliser cette fonction parce qu'utilisée dans des contextes modifiant différemment l'état, Pip-MPU retarde le checkpoint une fois un état consistent retrouvé. Ainsi, Pip-MPU divise ses propriétés de cohérence selon les checkpoints, actuellement en deux sous-ensembles.

Par ailleurs, Pip-MPU effectue quand même une propagation des listes utilisées dans les propriétés de l'invariant d'isolation tout comme Pip (MMU). Il est beaucoup plus compliqué de retrouver ces modifications plus tard sur un état qui a changé de multiples fois.

Tout particulièrement, la preuve d'un service de modification d'état se déroule de façon itérative dans Pip-MPU. L'esprit de la technique est de rapidement traverser l'ensemble de la preuve en propageant l'état modifié mais en ne prouvant pas les éléments de preuve difficile, avant de revenir dessus puis d'augmenter la complexité de la preuve à chaque itération. C'est une approche de détection rapide des bugs qui pourraient sinon invalider les propriétés de sécurité.

Tout d'abord, il faut effectuer une analyse préliminaire pour comprendre ce que fait le service et pour avoir une idée précise de l'état de la mémoire à sa terminaison, notamment celles des listes impliquées dans les propriétés. Il est alors possible de réaliser une preuve informelle des propriétés de sécurité pour éviter les bugs les plus évidents.

Ensuite se tient la phase de vérification simple, contenant que des instructions de lecture facilement prouvables en appliquant un motif de preuve connu. Les primitives bas-niveau qui ne disposent pas encore de preuve sont prouvées à ce moment-là car facile à faire.

Puis il y a la rencontre avec les instructions d'écriture. Cela se fait en plusieurs passes. D'abord, nous écrivons sur papier ou un tableau ces instructions et dégageons les propriétés attendues à chaque instruction. Nous pouvons ensuite appliquer le motif

de preuve, développons les preuves des primitives bas-niveau de lecture et d'écriture lorsqu'il le faut et simulons la preuve des fonctions internes (les *admit* en Coq).

A la suite de quoi nous prouvons les propriétés de sécurité, de façon isolée du reste. La preuve se fait en supposant toutes les propriétés de cohérence vraies et avec l'état modifié réel qui a été propagé au fur et à mesure des modifications. La preuve informelle est alors formellement exprimée et vérifiée. Si des propriétés manquent, alors il faut retrouver les instructions qui devraient les révéler et prouver que ces propriétés y sont bien présentes, puis les propager, sinon cela peut être révélateur d'un bug.

Une fois les propriétés de sécurité prouvées, il nous faut réitérer depuis la première instruction de modification pour prouver les obligations de preuve laissées de côté. Cela se fait en deux étapes : d'abord par la preuve des propriétés de cohérence simples, sans listes, tout du long du chemin de preuve déjà emprunté ; puis dans un second temps, les propriétés de cohérence plus difficiles. Les pré- et post-conditions des fonctions doivent être localement adaptées à leur contexte. C'est alors le moment d'identifier les checkpoints et prouver l'ensemble des propriétés de cohérence à ces instants. Dans un dernier effort, les propriétés propagées du début jusqu'à l'instant de preuve des propriétés de sécurité doivent se connecter.

A.4.3 Stratégie globale de conduite de preuve

Nous prenons maintenant un peu plus de hauteur sur le processus de preuve. En effet, nous avons supposé jusqu'à maintenant que nous connaissions l'ensemble des propriétés à prouver, et nous avons des techniques pour les vérifier formellement. Cependant, cela pose la question de leur découverte en premier lieu. Développer de multiples propriétés de cohérence qu'il faudra prouver mais qui ne seraient jamais utilisées pour la preuve des propriétés de sécurité s'avèrerait particulièrement inefficace. A la place, nous proposons pour Pip-MPU les **stratégies d'exploration horizontale et verticale**.

Le but de l'exploration horizontale est de couvrir un maximum de services pour révéler le plus de propriétés de cohérence possibles. Nous avons montré, déjà confirmé avant par Pip (MMU), que la phase de vérification des paramètres était plus facile à passer que la phase de modification. Ainsi, l'exploration horizontale va jusqu'à atteindre la phase de modification dans **tous** les services et fait ressortir le maximum de propriétés sur des entrées différentes. Le but est de prendre les devants sur les preuves futures des autres services qui reposeront sur ces propriétés et nous aurons alors à repasser moins fréquemment (ou plus du tout) sur la preuve d'un ancien service pour prouver des nouvelles propriétés. L'exploration verticale, quant à elle, sélectionne un service à prouver et va jusqu'au bout de la preuve de ce service avec la méthode décrite précédemment.

A.5 Suivi du développement de la preuve et optimisation

Le développement des preuves de Pip-MPU a soulevé le problème de communiquer mes avancements à ma hiérarchie non-experte. J'ai alors proposé de nouvelles métriques pour concevoir un tableau de bord relatant l'avancement à différents instants du processus, ainsi que la démonstration que nous suivions le chemin de preuve idéal.

A.5.1 Métriques de preuve existantes

Je me suis intéressé à deux catégories principales de métriques : sur les preuves en elles-mêmes et sur le processus de preuve. Ce dernier étant fortement lié au premier.

Les métriques sur les preuves communément affichées dans les projets de vérification formelle [85, 109, 102] sont le nombre de lignes de preuve, le nombre de théorèmes et lemmes, le nombre d'invariants, le nombre de bugs découverts et le taux de couverture de code vérifié. Beaucoup de ces métriques ont déjà été présentées avant, ainsi que des métriques additionnelles comme l'utilisation de la mémoire vive ou le nombre de tactiques Coq. Certains projets voient des relations entre ces métriques, comme une relation linéaire entre l'effort de preuve et la taille de la preuve ou encore une relation quadratique entre la taille d'une propriété et son script de preuve, mais sont énormément liées au style de preuve de chaque projet [167, 122].

Pour parler du processus de preuve, nous définissons d'abord l'effort de preuve :

Definition A.5.1 (Effort de preuve)**.** L'effort de preuve est un indicateur de la quantité de travail déployée pour réaliser une preuve (effort humain).

Dans les projets de vérification formelle, l'effort de preuve est d'habitude exprimé en personne-temps (jours, mois, années). Le processus de preuve est impacté par l'approche de preuve, comme le raffinement à différents étages et granularité, qui mène à des métriques qui ont des sens différents selon le projet [5]. D'autres travaux [21] ont tentés d'identifier des caractéristiques de preuve pour quantifier un effort de preuve (nombre de théorèmes pondéré par complexité, taille du théorème et nombre de dépendances) ou pour détecter des erreurs potentielles (profondeur de l'arbre de dépendance et nombre de théorèmes imbriqués qui rendent plus difficiles la maintenance, score de similarité entre des théorèmes identifiant une mauvaise conception).

A.5.2 Relation entre code et preuve

Nous mettons en relation les **éléments de preuve** (théorème ou lemmes et preuves associées) avec les **éléments de code** (fonctions, modules, instructions). Tout comme les éléments de code sont assemblés pour construire un service, les éléments de preuve sont assemblés lors de la preuve. Dans Pip-MPU, l'approche est de faire correspondre à tout élément de code un élément de preuve. Ainsi, le nombre d'éléments de preuve minimum est connu à l'avance.

Theorem A.5.1 (Nombre minimum d'éléments de preuve). Pour l'ensemble des éléments de code composés par le sous-ensemble de fonctions $F = F_1, F_2, ..., F_f$ et le sous-ensemble de primitives $P = P_1, P_2, ..., P_p$, et pour l'ensemble des éléments de preuve correspondants $PE = PE_1, PE_2, ..., PE_l$, alors

$$|F| + |P| <= |PE|$$

De plus, nous avons une correspondance entre éléments de code et éléments de preuve, alors le graphe de dépendances de chacun est identique vu depuis un haut niveau d'abstraction (les éléments principaux). Le graphe de dépendances est un polyarbre, un type d'abre orienté acyclique dont le graphe non orienté sous-jacent est un arbre. Il trace les routines internes de chaque service et s'arrête au-dessus du niveau des primitives, soit aux fonctions qui disposent d'au moins deux instructions.

Definition A.5.2 (Graphe de dépendance)**.** Le graphe des dépendances fonctionnelles, respectivement de preuve, est défini comme l'ensemble $G_{fonction} = (F, DC)$, respectivement $G_{preuve} = (PE, DP)$, où l'ensemble des fonctions $F = F_1, ..., F_n$ sont les noeuds de $G_{fonction}$, respectivement où l'ensemble des éléments de preuve principaux $PE = PE_1, ..., PE_n$ sont les noeuds dans G_{preuve}, et où l'ensemble des dépendances DC sont les arrêtes de $G_{fonction}$, respectivement DP dans G_{preuve}.

La réutilisabilité des fonctions bénéficie également la preuve dont l'effort de preuve est réduit d'autant. Si la preuve ne suit pas la modularité du code, alors l'effort de preuve grandit d'autant de composants non réutilisés mais qui aurait pu. Nous sommes intéressés par quantifier ce taux de réutilisabilité potentielle par les preuves.

Pour cela, nous définissons d'abord le sous-graphe des éléments principaux réutilisés et le sous-graphe complet des éléments réutilisés.

Definition A.5.3 (Sous-graphe des éléments principaux réutilisés)**.** Les éléments principaux réutilisés sont les noeuds du graphe de dépendance G dont le degré entrant est supérieur à 1. Le sous-graphe des éléments principaux réutilisés est alors $R = \{G' \in G | \forall E_i \in G[E], deg^-(E_i) > 1\}$.

Definition A.5.4 (Sous-graphe complet des éléments réutilisés)**.** Toutes les routines internes des éléments réutilisés sont bien entendu également réutilisées, mais pourraient ne pas apparaître dans R si leur degré est 1. Le sous-graphe complet des éléments réutilisés est alors R augmenté de tous les éléments des sous-arbres $S = S_1, ..., S_s$, $s \in \mathbb{N}$, enracinés dans un élement principal réutilisé, donc $G_{élément\_réutilisé} = \{G' \in G | \forall G_i \in G, \forall R_j \in R, \forall S_k \in S_{R_{\lfloor j}}, G_i[E] \in S_k \wedge deg^-(G_i[E]) = 1\} =$
$(E_{élément\_réutilisé}, A_{élément\_réutilisé})$.

Definition A.5.5 (Ratio d'éléments réutilisés)**.** Le ratio d'éléments réutilisés RR est le ratio du nombre d'éléments réutilisés $E_{élément\_réutilisé}$ sur le nombre total d'éléments dans le graphe de dépendance G :

$$RR = \frac{|E_{élément\_réutilisé}|}{|E|}$$

A noter que les noeuds principaux des services ne sont jamais dans $E_{élément\_réutilisé}$ étant donné qu'ils n'ont pas d'arrêtes entrantes.

Appliqué à Pip-MPU, il y a que la modularité des preuves reprend celle du code, donc $G_{fonction} = G_{preuve} = G$. Afin de rendre le propos plus clair et lisible, seuls les éléments comportant des instructions d'écriture ont été reportés dans la Figure A.4, cependant le graphe complet de dépendance devrait inclure l'ensemble des fonctions y compris celles avec uniquement des instructions de lecture. Sur cette figure, les éléments réutilisés sont indiqués en bleus. Plus il y a de bleu, plus il y a d'éléments réutilisés donc.

Le ratio d'éléments réutilisés pour Pip-MPU considérant ce graphe de dépendance simplifié est de :

$$\frac{10}{36} \approx 27.7\%$$

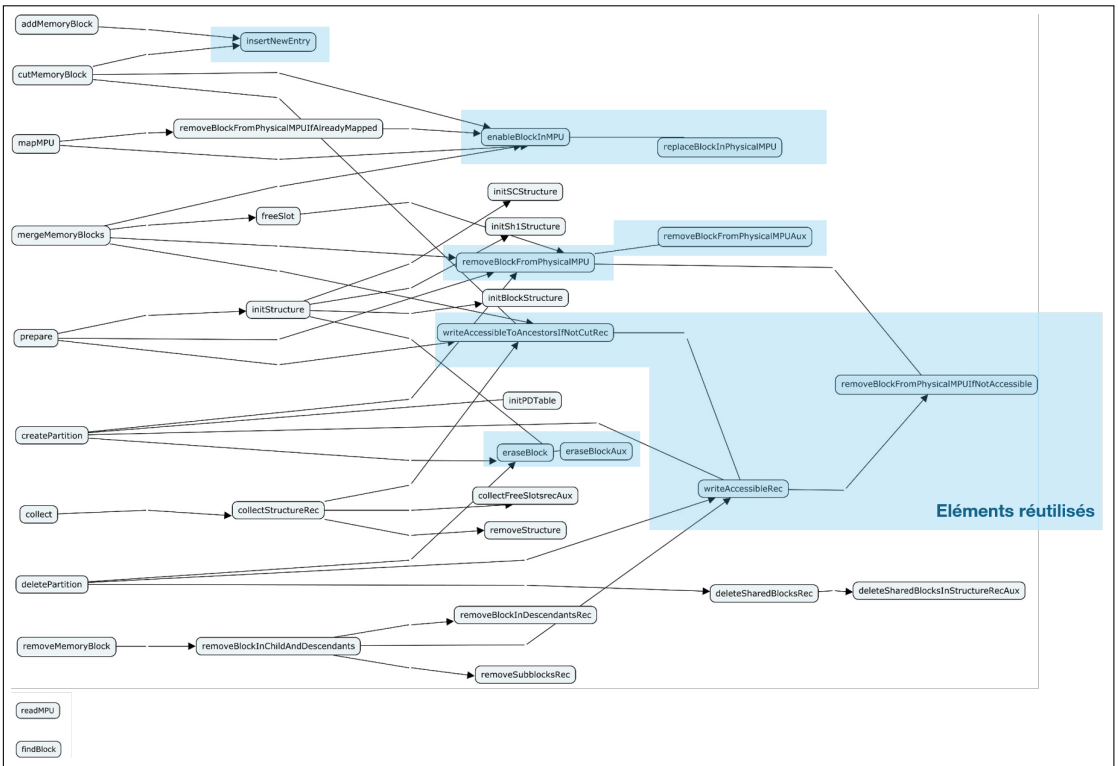


Figure A.4: Graphe de dépendance de Pip-MPU. Les points d'entrée des services sont à gauche. Les éléments réutilisés sont dans les boîtes bleues.

A.5.3 Réutilisation effective

La précédente métrique n'inclut pas l'utilisation effective des éléments réutilisés. Autrement dit, la fréquence de réutilisation.

Pour calculer ce taux de réutilisation effectif, nous définissons trois graphes : le graphe de dépendance déroulé, le sous-graphe de dépendance délesté des éléments non réutilisés et le sous-graphe délesté des éléments réutilisés.

Definition A.5.6 (Graphe de dépendance déroulé)**.** Le graphe de dépendance déroulé est défini comme $G_{déroule} = (F', A')$ qui est G sans aucune modularité avec F' et A' les nouveaux ensembles de fonctions et de dépendances du graphe déroulé. C'est comme si nous suivions chaque chemin de dépendance depuis les noeuds racine des services en copiant-collant tous les éléments réutilisés à chaque passage, soit le pire pour du développement de code ou de preuve et leur maintenance. Le nombre total d'éléments à développer est $|F'|$. $|F'|$ est égal au nombre total d'éléments de chaque sous-arbre $SG' = SG'_1, ..., SG'_s \subseteq G$ enracinés dans les noeuds des services, soit $|F'| = \sum_{i=1}^{s} |SG'_i|$.

Definition A.5.7 (Sous-graphe de dépendance délesté des éléments non réutilisés)**.** Le sous-graphe de dépendance délesté des éléments non réutilisés est le sous-graphe de dépendance de code $G_{délesté} = (F'', A'')$ qui est G délesté des éléments réutilisés avec F'' and A'' les nouveaux ensembles de fonctions et de dépendances du sous-graphe délesté. C'est-à-dire que nous sommes intéressés par $G_{délesté} = G \setminus G_{élément\_réutilisé} = (F'', A'')$.

Nous pouvons désormais définir la réutilisation effective.

Definition A.5.8 (Réutilisation effective)**.** Nous considérons les polyarbres $G_{élément\_réutilisé}$, $G_{délesté}$ et tous ses sous-arbres $SG'' = SG''_1, ..., SG''_r \subseteq G_{délesté} \subseteq G$ enracinés dans les noeuds de services ($r \in \mathbb{N}$), et $G_{déroule}$ avec tous ses sous-arbres $SG' = SG'_1, ..., SG'_s \subseteq G$ enracinés dans les noeuds de service ($s \in \mathbb{N}$). Puisque les sous-arbres enracinés dépendent des mêmes noeuds de service, alors $r = s$.

Ainsi, la réutilisation effective ER est définie par :

$$ER = \frac{|G_{élément\_réutilisé}| + \sum_{i=1}^{r} |SG''_i|}{\sum_{i=1}^{r} |SG'_i|} = \frac{|G_{élément\_réutilisé}| + \sum_{i=1}^{r} |SG''_i|}{|F'|}$$

ER donne le taux d'éléments à prouver avec réutilisation par rapport à sans réutilisation, et $1 - ER$ donne le taux d'éléments à prouver évité grâce à la réutilisation.

Par la suite, nous mentionnerons le diviseur de ER comme E et son dividende par U, soit $ER = \frac{U}{E}$.

L'ajout de $|G_{élément\_réutilisé}|$ correspond au coût unique de vérification des éléments réutilisés qu'il faut connecter au reste de la preuve induisant un certain coût minimal.

La réutilisation effective dans Pip-MPU est calculée pour chaque service de gestion mémoire dans le Tableau A.9.

A.5.4 Différence de réutilisation

Dans certaines situations, les graphes de dépendance entre code et preuve pourraient ne pas correspondre, par exemple à cause de lemmes généralistes sur les fonctions qui sont difficiles à implémenter ou parce que l'effort de preuve n'est pas rentable sur de

Service ($r = 11$)	$\lvert SG_i'' \rvert$	$\lvert SG' \rvert$
addMemoryBlock	1	2
cutMemoryBlock	1	7
mapMPU	2	6
mergeMemoryBlocks	2	13
prepare	5	13
createPartition	2	9
collect	4	7
deletePartition	3	7
removeMemoryBlock	4	6
readMPU	1	1
findBlock	1	1
Total	26	72
U = $\lvert G_{élément\_réutilisé} \rvert$ + 26 = 10+26 = 36		
E = |F'| = 72		
Réutilisation effective : $\frac{U}{E} = \frac{36}{72} = 50\%$		

Table A.9: Réutilisation effective dans Pip-MPU.

petites fonctions. Il y a alors une différence entre les deux graphes qui se remarque par la différence des réutilisations dans le code des réutilisations dans la preuve.

Definition A.5.9 (Différence de réutilisation code-preuve)**.**

$$[Réutilisation\ code] - [Réutilisation\ preuve]$$

Plus cette différence est importante, plus cela indique des opportunités ratées d'optimisation, de facilitation de la conception des preuves, des scripts de preuve dupliqués ou l'utilisation de lemmes additionnels inutiles.

Dans Pip-MPU, cette différence est nulle et montre que le processus de preuve a utilisé le plein potentiel de la modularité du code pour ne pas avoir d'efforts de preuve supplémentaire. En effet, dans le cas de Pip-MPU, le développeur est plus intéressé par réduire son effort de preuve plutôt que d'avoir la preuve la plus compacte et la plus élégante moyennant d'importants efforts.

A.5.5 Complexité de la preuve : analyse fine des propriétés et des impacts du code sur la preuve

Nous prenons maintenant du recul sur la preuve, et souhaitons estimer l'effort de preuve global afin de pouvoir communiquer sur l'avancement de la preuve et décider d'une stratégie qui prendrait en compte cet effort estimé. L'intuition ici est qu'une instruction donnée aura un certain impact sur certaines propriétés mais peu sur d'autres et que l'on peut le quantifier. Cette mesure combinée à la complexité intrinsèque des propriétés permet de dériver une complexité globale à l'échelle du service. Les différences de complexité nous indiquent un effort de preuve plus ou moins important qui sera notre indicateur pour développer la stratégie globale.

		Score d'impact sur les propriétés de cohérence	Score d'impact sur les propriétés de sécurité
Type	PDT type	14 + 5*	3*
	BE type	15 + 10*	3*
	SHE type	6 + 2*	3*
	SCE type	3	0
	PADDR type	7 + 11*	3*
Champ	pdentry.(firstfreeslot)	2 + 3*	0
	pdentry.(structure)	3 + 9*	3*
	pdentry.(nbfreeslots)	1 + 3*	0
	pdentry.(parent)	2	0
	pdentry.(MPU)	1	0
	blockentry.(blockindex)	5	0
	blockentry.(blockrange).(startAddr)	3 + 5*	3*
	blockentry.(blockrange).(endAddr)	1 + 7*	3*
	blockentry.(present)	5 + 7*	3*
	blockentry.(accessible)	4 + 2*	1*
	blockentry.(read)	2	0
	blockentry.(write)	2	0
	blockentry.(execute)	2	0
	sh1entry.(PDflag)	5 + 1*	3*
	sh1entry.(PDchild)	3	0
	sh1entry.(inChildLocation)	3	0
	scentry.(origin)	2	0
	scentry.(next)	2	0

Table A.10: Illustration des impacts de modifications de types et des entrées sur les propriétés de Pip-MPU.

Nous avons déjà vu plus tôt que l'impact d'une instruction de lecture sur la preuve est quasiment nul, en tous cas négligeable face à une instruction d'écriture.

Pour les instructions d'écriture, l'impact sur la preuve dépend de la modification. Par exemple, dans Pip-MPU, l'impact d'une modification des permissions d'accès ne sont pas critiques pour l'invariant d'isolation : cela n'impacte pas du tout les propriétés de sécurité. Au contraire, un drapeau PDflag d'une entrée Shadow 1 qui est levé bouleverse l'arbre de partitionnement qui contiendra un élement en plus, impactant là les propriétés de sécurité mais également des propriétés de cohérence dont $PDTIFPDflag$.

Nous analysons alors l'ensemble des propriétés de l'invariant d'isolation en fonction des modifications des entrées. A chaque fois que cela impacte une propriété, nous augmentons de un point le score global d'impact pour ce champ d'entrée modifié. Une exception est faite sur les listes. En effet, nous avons déjà discuté du développement de lemmes spécifiques pour leur propagation instruction par instruction ce qui représente un coût important mais une seule fois. Nous notons alors leur impact d'une '*' et devons les additionner à part. Nous obtenons le Tableau récapitulatif A.10.

Nous pouvons en déduire que toutes les modifications concernant les drapeaux *present* et $PDflag$, amplifiées par les scores d'impact sur les types BE et PDT, sont les plus importantes pour les propriétés et par extension les preuves. Au contraire, comme attendu, les permissions d'accès dérangent peu les propriétés.

Du point de vue de la fonction ou du service, le score d'impact dépend de l'impact des instructions les composant.

Definition A.5.10 (Score d'impact d'une fonction)**.** Pour l'ensemble des propriétés P à prouver ($|P| = p$), les instructions I de la fonction F ($|I| = n$) et la matrice liant les deux ensembles $M = (P, I)$, le score d'impact IS de F est défini par

$$IS = \sum_{i=1,j=1}^{n,p} M[P_j, I_i]$$

De façon similaire, le score d'impact d'un service est la somme de tous les scores d'impact de l'ensemble des éléments de preuve. Les propriétés à prouver incluent toutes les propriétés de l'invariant d'isolation, dont les propriétés de sécurité.

Definition A.5.11 (Score d'impact du service)**.** Pour un service S composé de l'ensemble des éléments de preuve PE ($|PE| = pe$) dont les scores d'impact sont contenus dans l'ensemble $ISPE$, et ayant les instructions I ($|I| = n$) dans la fonction principale du service, les propriétés P de l'invariant d'isolation ($|P| = p$), et considérant la matrice reliant les deux ensembles $M = (P, I)$, alors le score d'impact ISS de S est défini par

$$ISS = \sum_{i=1,j=1}^{n,p} M[P_j, I_i] + \sum_{i=1}^{pe} ISPE_i + C$$

$C = 1$ est un score d'impact additionnel ajouté à chaque élement de preuve puisqu'il y a toujours des ajustements locaux à fournir pour insérer ces éléments.

Nous illustrons ce score avec le service `addMemoryBlock`. D'abord nous calculons le score d'impact de la fonction interne `insertNewEntry`. Celle-ci contient 14 instructions, dont 10 sont des instructions de modification. En additionnant les scores d'impact de chacune de ces instructions selon le Tableau A.10, nous obtenons un score d'impact total de (2+3*)+(1+3*)+(3+5*+3*)+(1+7*+3*)+(5+7*+3*)+(4+2*+1*)+2+2+2+2=24+37*. Puis nous calculons le score d'impact de la fonction principale du service. Des 29 instructions, seules 2 sont des instructions de modifications. Le score d'impact pour cette fonction, selon le même tableau, est de 3+3=6. Enfin, nous additionnons le tout et obtenons un score d'impact final de 30+37*. Ce score est plus important qu'un service sans instructions de modifications comme `findBlock` qui a un score d'impact de 0.

Le score d'impact des instructions est insuffisant pour décrire la complexité de la preuve. En effet, les propriétés ont aussi leur rôle à jouer.

J'ai retenu quatre caractéristiques qui augmentent la complexité de la preuve d'une propriété : 1) la taille de la propriété (le nombre de sous-propositions) qui influence directement le contexte de preuve à adapter et transformer, 2) le nombre de listes et non leur présence car nous avons vu que l'impact pouvait être diminué en développant

des lemmes intermédiaires de propagation des listes par instructions, 3) le nombre de variables en jeu car cela ajoute des combinaisons de cas à traiter, et 4) le nombre sous-propositions finales puisque c'est le nombre final de preuve à fournir pour résoudre la preuve de la propriété.

Selon ces caractéristiques, nous obtenons la matrice de complexité des propriétés de Pip-MPU représentée dans le Tableau A.11. Ce tableau montre de grandes disparités de complexité, avec les plus fortes complexités remarquées dans les propriétés *freeSlotsListIsFreeSlot*, *DisjointFreeSlotsLists*,*accessibleChildPaddrIsAccessibleIntoParent* et *sharedBlockPointsToChild*.

La complexité de preuve reflète la difficulté globale qui requiert des capacités de résolutions de problèmes. C'est une combinaison des scores d'impact des instructions avec les complexités des propriétés.

Definition A.5.12 (Complexité de preuve d'une instruction)**.** La complexité de preuve PC d'une instruction I est définie comme la multiplication de la colonne correspondant dans la matrice d'impact des instructions IIM avec la colonne calculant le total de la matrice de complexité de propriété PCM :

$$PC_I = IIM[I] * PCM[Total]$$

Au niveau d'une fonction, la complexité de la preuve mesure la difficulté apportée par chaque instruction pour prouver les propriétés. Une exception est faite pour le type d'entrée qui est modifié. En effet, les types sont inclus dans la modification d'un champ, dans le sens où le type initial doit être connu avant d'effectuer la modification. Cependant, du point de vue de la preuve, l'intérêt n'est pas dans le nombre de modifications de types. En effet, si plusieurs instructions modifient la même adresse avec le même type, alors la propriété de type nécessaire pour la première instruction suffit pour les suivantes. Ainsi, l'impact d'une modification de type doit être apprécié du point de vue de la fonction et du nombre d'adresses modifiées.

Definition A.5.13 (Complexité de preuve de type)**.** Supposons l'association $M : I \rightarrow (T, A)$ associant les instructions I à leur type de l'ensemble des types T et l'adresse mémoire modifiée de l'ensemble des adresses mémoire A. La complexité de preuve du type pour la fonction F nommée $TFPC$ est définie comme l'ensemble de tuples $TA = \{(t, a) | t \in T \wedge a \in A\}$ de taille s correspondant à M appliquée à toutes les instructions de F multipliée par les scores d'impact de type correspondants TIS:

$$TFPC_F = \sum_{i=1}^{s} TIS[TA_i[t]]$$

Par exemple, la fonction `insertNewEntry` manipule trois types différents : PDT, BE et SCE. Pour chaque type, seule une adresse mémoire est associée, donc trois adresses différentes. Ainsi, la part de la complexité pour la fonction due à la modification de type est $TIS[PDT] + TIS[BE] + TIS[SCE]$ =14 + 8* + 15 + 13* + 3 = 32 + 24*.

Finalement, nous définissons la complexité de preuve d'une fonction prenant en compte les instructions et les types utilisés dans la fonction.

	Properties	Taille de la propriété	Nombre de listes	Nombre de variables	Nombre de propositions finales	Total
cohérence	nullAddrExists	1	0	0	1	2
	wellFormedFstShadowIfBlockEntry	2	0	1	1	4
	PDTIfPDFlag	2	0	2	4	8
	AccessibleNoPDFlag	4	0	2	1	7
	FirstFreeSlotPointerIsBEAndFreeSlot	2	0	2	2	6
	multiplexerIsPDT	1	0	0	1	2
	currentPartitionInPartitionsList	1	1	0	1	3
	wellFormedShadowCutIfBlockEntry	2	0	1	2	5
	BlocksRangeFromKernelStartIsBE	3	0	2	1	6
	KernelStructureStartFromBlockEntryAddrIsKS	3	0	2	1	6
	sh1InChildLocationIsBE	3	0	2	1	6
	StructurePointerIsKS	2	0	2	1	5
	NextKSIsKS	5	0	3	1	9
	NextKSOffsetIsPADDR	3	0	2	1	6
	NoDupInFreeSlotsList	2	1	2	3	8
	freeSlotsListIsFreeSlot	6	1	4	1	12
	DisjointFreeSlotsLists	4	2	2	5	13
	inclFreeSlotsBlockEntries	3	2	1	1	7
	DisjointKSEntries	4	2	2	1	9
	noDupPartitionTree	1	1	1	1	4
	isParent	3	2	2	1	8
	isChild	3	2	2	1	8
	accessibleChildPaddrIsAccessibleIntoParent	4	4	3	1	12
	noDupKSEntriesList	2	1	1	1	5
	noDupMappedBlocksList	2	1	1	1	5
	noDupUsedPaddrList	2	1	1	1	5
	sharedBlockPointsToChild	7	5	5	2	19
	MPUInAccessibleBlocks	3	1	2	1	7
Securité	Partage Vertical	3	4	2	1	10
	Isolation Horizontale	5	5	3	1	14
	Isolation du noyau	3	4	2	1	10

Table A.11: Complexités des propriétés de Pip-MPU.

Definition A.5.14 (Complexité de preuve d'une fonction)**.** La complexité de preuve PC d'une fonction F composée de n instructions est définie comme la somme de la complexité de chacune de ses instructions augmentée de la complexité due aux types utilisés dans F :

$$PC_F = TFPC_F + \sum_{1}^{n} PC_i$$

Par example, prenons le service `addMemoryBlock`. Les instructions de modifications de la fonction principale du service sont `writeSh1PDChildFromBlockEntryAddr`, qui modifie le champ PDchild de l'entrée Shadow 1, et `writeSh1InChildLocationFromBlock EntryAddr`, qui modifie le champ InChildLocation de la même entrée Shadow 1. Ces instructions ont respectivement une complexité de $PC_{sh1entry.pdchild} = 6{*}1{+}12{*}1{+}19{*}1 = 37$ et de $PC_{sh1entry.inchildlocation} = 6*1+6*1+12*1 = 24$.

La fonction `insertNewEntry` a une complexité de preuve de 228 + 58* comme calculée dans le Tableau A.12 (*cf.* corps principal du document en anglais pour le calcul détaillé), si nous considérons toutes les propriétés de cohérence. La complexité de preuve du service `addMemoryBlock`'s est donc de $PC_{addMemoryBlock} = 37+24+228+58* = 289+58*$.

Enfin, nous pouvons définir l'effort de preuve global. Il y a de multiples façons pour le calculer. Par exemple, il peut rétrospectivement être calculé selon les personne-mois ou personne-jours dédiés à l'activité de vérification formelle qui sont d'excellents indicateurs standards pouvant être utilisés pour faire un équivalent monétaire. Mais cela ne prend pas en compte les difficultés de la tâche ni l'expérience des développeurs de preuves.

Definition A.5.15 (Effort de preuve estimé)**.** L'effort de preuve estimé EPE repose sur la complexité de preuve PC des n ($n \in \mathbb{N}$) éléments de preuve à prouver et définie comme :

$$EPE = n + \sum_{i=1}^{n} PC_i$$

Avec l'EPE, il est possible d'estimer à l'avance l'effort de preuve à fournir en ayant une idée de la complexité de la tâche. L'EPE peut être adapté à tous les projets ayant une définition précise de la complexité de preuve.

A.5.6 Stratégie du chemin de preuve au moindre effort

Précédemment, nous avions suivi globalement les explorations horizontale et verticale pour rapidement identifier les propriétés de cohérence et tester notre processus de preuve. Est-il possible désormais d'optimiser le chemin de preuve des services restants ? Autrement dit, nous disposons maintenant de métriques permettant d'estimer un effort de preuve, mais y a-t-il un ordre meilleur qu'un autre pour prouver les éléments de preuve ? Nous adoptons dans Pip-MPU une stratégie pour minimiser l'effort de preuve courant tout en minimisant également l'effort de preuve restant.

Pour Pip-MPU, cela se résume à trouver l'ordre optimal de réduction de l'effort de preuve tout en validant les services un par un pour observer l'avancement de la preuve à l'échelle de l'IPA. Par ailleurs, c'est à la fin du service que l'invariant d'isolation est

Complexité de preuve des instructions	
Instructions	Complexité
writePDFirstFreeSlotPointer	13 + 3*
writePDNbFreeSlots	7 + 3*
writeBlockStartFromBlockEntryAddr	26 + 8*
writeBlockEndFromBlockEntryAddr	7 + 10*
writeBlockAccessibleFromBlockEntryAddr	33 + 3*
writeBlockPresentFromBlockEntryAddr	35 +10*
writeBlockRFromBlockEntryAddr	18
writeBlockWFromBlockEntryAddr	18
writeBlockXFromBlockEntryAddr	18
writeSCOriginFromBlockEntryAddr	18
Sous-total	193 + 37*
Complexité de type	
Nombre de modifications de type PDT à des adresses différentes : 1	14+8*
Nombre de modifications de type BE à des adresses différentes : 1	15+13*
Nombre de modifications de type SHE à des adresses différentes : 0	0
Nombre de modifications de type SCE à des adresses différentes : 1	3
Nombre de modifications de type PADDR à des adresses différentes : 0	0
Sous-total	35 + 21*
Total	228 + 58*

Table A.12: Illustration du calcul de complexité de preuve de la fonction `insertNewEntry`.

prouvé, et donc qui nous donne confiance dans notre conception. J'ai développé pour cela l'Algorithme 4 où l'objectif est de minimiser l'effort courant tout en maximisant la couverture de code globalement grâce aux éléments réutilisés. Ainsi, par une réutilisation effective élevée A.5.3, cela nous donne plus de flexibilité pour choisir la meilleure stratégie de conduite de preuve.

Algorithm 4 Algorithme de sélection de meilleur chemin par minimisation de l'effort de preuve.

```
1: ordre_services ← []
2: effort_preuve_courant ← []
3: effort_preuve_suivant ← []
4: repeat
5:    for service dans services do
6:       chemin_preuve ← get_chemin_preuve(service) { Récupère le chemin de preuve
         de service}
7:       effort_preuve_courant[service] ← get_effort_preuve_courant(chemin_preuve){
         Calcul de l'effort de preuve courant }
8:       effort_preuve_suivant[service] ← get_effort_preuve_suivant(chemin_preuve)
         { Calcul les bénéfices pour les autres services }
9:    end for
10:   effort_preuve_courant_min ← get_services_effort_preuve_courant_min
      (effort_preuve_courant) { Sélectionner les services par le minimum de la 1ère
      valeur}
11:   if taille(effort_preuve_courant_min) > 1 { si plusieurs services ont le même
      effort de preuve} then
12:      effort_preuve_suivant_max ← get_services_avec_effort_preuve_suivant_max
         (effort_preuve_courant_min, effort_preuve_suivant) { Prendre le maximum de
         la 2e valeur des services sélectionnés par la 1ère valeur}
13:      service_slectionn ← effort_preuve_suivant_max[1] { Toujours prendre le pre-
         mier service dans la list, même si plusieurs services ont le même score }
14:   else
15:      service_slectionn ← effort_preuve_courant_min[1]
16:   end if
17:   ordre_services.ajout(service_slectionn) { Ajouter le service sélectionné dans la
      liste des services ordonnés}
18:   enlever_chemin_preuve(service_slectionn) { Enlever tous les éléments de preuve
      du chemin de preuve du service sélectionné }
19: until tous les services dans ordre_services
20: return ordre_services{ Retourner l'ordre des services }
```

Nous illustrons l'application de l'Algoritme 4 à Pip-MPU dans le Tableau A.13.

Pour simplifier les calculs, nous approximons l'effort de preuve par le nombre d'éléments de preuve sur chaque chemin. L'effort de preuve suivant est calculé comme la somme de tous les arcs entrants d'un élément sur le chemin moins l'arc entrant dû au chemin lui-même (on ne garde que les arcs entrants additionnels). En effet, comme décrit précédemment, les arcs entrants d'un chemin de dépendance représentent les

Iteration	addMemoryBlock	cutMemoryBlock	mapMPU	mergeMemoryBlocks	prepare	createPartition	collect	deletePartition	removeMemoryBlock	readMPU	findBlock
1	2/1	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	1/0°	1/0
2	2/1	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	0/0	1/0°
3	2/1°	7/11	4/3	9/10	12/8	8/9	7/7	7/6	6/4	-	0/0
4	0/0	6/10	4/3°	9/10	12/8	8/9	7/7	7/6	6/4	-	-
5	-	4/7°	0/0	6/1	12/8	8/9	7/7	7/6	6/4	-	-
6	-	0/0	-	4/3°	7/3	6/4	4/0	5/1	4/0	-	-
7	-	-	-	0/0	5/0	4/1°	4/0	5/1	4/0	-	-
8	-	-	-	-	5/0	0/0	4/0	3/0°	4/0	-	-
9	-	-	-	-	5/0	-	4/0°	0/0	4/0	-	-
10	-	-	-	-	5/0	-	0/0	-	4/0°	-	-
11	-	-	-	-	5/0°	-	-	-	0/0	-	-
12	-	-	-	-	0/0	-	-	-	-	-	-

Table A.13: Itérations de l'Algorithme 4 de sélection du meilleur chemin de preuve par minimisation de l'effort de preuve appliqué à Pip-MPU. Chaque itération donne un nouveau score qui dépend des services sélectionnés précédemment. Le score est donné comme un tuple ($effort\_preuve\_courant$ / $effort\_preuve\_suivant$). Un service marqué par '°' dans une itération indique que ce service a été sélectionné à ce moment. L'ordre final des services est : [readMPU, findBlock, addMemoryBlock, mapMPU, cutMemoryBlock, mergeMemoryBlocks, createPartition, deletePartition, collect, removeMemoryBlock, prepare]

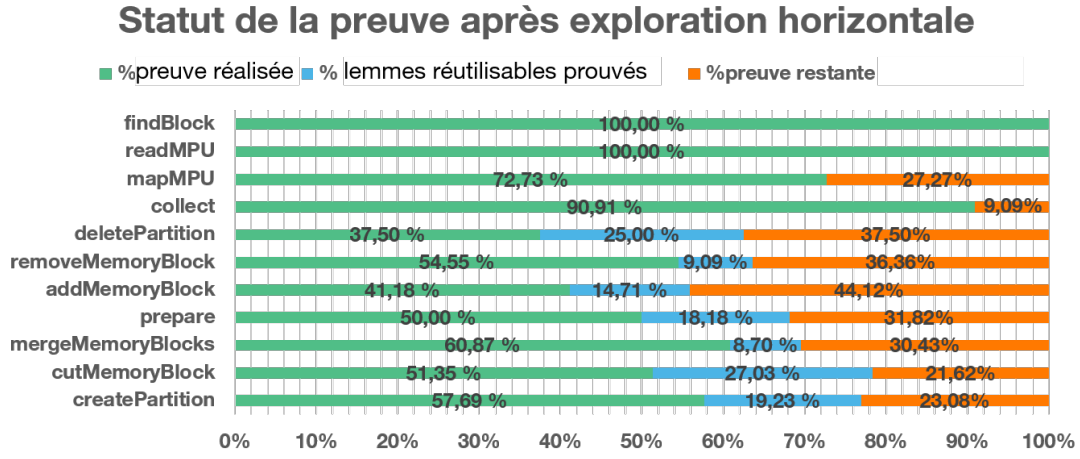


Figure A.5: Etat de la preuve après l'exploration horizontale estimée par nombre d'éléments de preuve et non par complexité.

lemmes réutilisables qui bénéficient aussi à d'autres services. L'exemple pourrait être affiné par les métriques d'effort de preuve présentés précédemment, qui dépendent des complexités de preuve. Dans ce cas, l'algorithme aurait sélectionné `removeMemoryBlock` plutôt que `collect` à l'itération 10.

A.5.7 Tableau de bord de preuve

Le tableau de bord de preuve est un outil dynamique qui donne le statut d'avancement de la preuve. Il modifie sa granularité en fonction des progrès sur la preuve.

Au début du développement des preuves, lors de la phase d'exploration horizontale, le statut est estimé par l'endroit où se situe le curseur de preuve par rapport au nombre d'instructions global, comme illustré dans la Figure A.5. Les instructions restantes sont soit des nouvelles primitives encore jusqu'alors non rencontrées, soit des instructions qui ont déjà été rencontrées et dont la preuve est facilitée (les lemmes réutilisables prouvés).

Puis, lors de l'exploration verticale, l'accent est mis sur un service. De nouveau, une première approximation est le nombre d'instructions. Cependant, nous pouvons désormais étudier la complexité de preuve et le score de complexité du service devient le nouvel objectif exposé dans le tableau de bord. Enfin, pour la dernière instruction, il ne reste plus qu'à prouver l'invariant d'isolation totalement. Le tableau de bord casse l'invariant propriété par propriété et montre la part de propriétés prouvées par rapport à l'ensemble complet des propriétés.

A.6 Conclusion

Pip-MPU atteint le plus haut niveau de confiance pour un système avec une sécurité ancrée dans le matériel par la MPU et vérifiée formellement par un assistant de preuve.

La thèse a démontré des contributions concernant un cadriciel mettant en place des espaces mémoire imbriqués sécurités et protégés par la MPU. Il a aussi été proposé une adaptation du noyau Pip pour les objets contraints nommé Pip-MPU qui satisfait toutes les exigences attendues pour l'adaptation. Par ailleurs, les propriétés de sécurité (d'isolation) de plusieurs services de Pip-MPU ont été formellement vérifiées par un développement de preuve au sein de l'assistant de preuve Coq. Enfin, de nouvelles techniques de preuves et métriques ont été proposées pour améliorer la vision extérieure de l'avancement de la vérification formelle à un public non-expert et choisir un chemin de preuve minimisant l'effort de preuve.

Toutes les contributions présentées sont publiées sous licence open-source.

Index

Contents

Design of a secure kernel for constrained objects

Abstract

This thesis invests the field of cybersecurity for small computer systems (embedded systems/-connected objects/low-end devices, of type microcontroller) and more precisely aims to bring strong memory isolation guarantees for tasks executing on them.

The heterogeneity and strong resource constraints (memory, computing power, energy) of constrained embedded systems require tailored solutions. The embedded software life cycle and the specific hardware platforms challenge us to reconsider the security schemes that leave open memory vulnerability issues, still prevailing today. Furthermore, the risk of vulnerability exploits elevates with the growing number of use cases (smart environments in general) implying increased complexity within these systems and with the burgeoning market of the Internet-of-Things (notably for remote update purposes). As a consequence, cyber attackers can take profit from these vulnerabilities to take remote control of these connected systems in a very scalable way.

In this context, the thesis proposes to design a kernel for constrained objects that is able to bring strong memory isolation guarantees. It studies the blend of high flexibility and strong security for this class of devices with the aim of security-by-design without functional loss. The thesis presents two main contributions dealing with software attacks on memory.

The first contribution is an Operating System (OS) kernel, named Pip-MPU, that offers a hardware-based isolation solution with a degree of flexibility outperforming current solutions. Pip-MPU is adapted from the Pip protokernel initially designed for high-end/general-purpose computers that are provided with more furnished and different hardware platforms than the ones of constrained objects. For that, the designed kernel is a complete refactoring of Pip and offers a security mechanism based on the Memory Protection Unit (MPU), a unit of the processor, which enables hardware-based access control on memory resources. Despite strong limitations due to the limited hardware platform, Pip-MPU is as flexible as its parent project Pip. With a code base size of less than 10 Kb and about 16% extra costs in terms of performance and energy consumption, Pip-MPU reduces the number of privileged operations by 99% and the attack surface of the accessible application memory by 98%.

The second contribution is the demonstration of strong isolation guarantees by the use of formal methods. Several kernel services have been proved against isolation by the use of the Coq Proof Assistant. The proved properties are Pip's security properties that enforce a strict memory isolation security model. To our knowledge, no state-of-the-art solution offering MPU-based isolation has been proved before. We develop novel proof conduct techniques and propose new metrics to follow the proof effort and evaluate the hypotheses supporting the proofs.

All the contributions developed in this thesis are publicly released under open-source licences.

Keywords: memory isolation, constrained objects, mpu, formal verification, coq, pip, security-by-design

Conception d'un noyau sécurisé pour objets contraints

Résumé

Cette thèse s'inscrit dans la thématique de la sécurité des petits systèmes informatiques (systèmes embarqués/objets connectés, de type microcontrôleur) et plus précisément vise à apporter des fortes garanties d'isolation mémoire pour les tâches qui s'y exécutent.

L'hétérogénéité et les fortes contraintes en ressources (espace mémoire, puissance de calcul, énergie) nécessitent la mise en place de solutions sur mesure pour les systèmes embarqués contraints. Le cycle de vie des logiciels embarqués et les architectures matérielles spécifiques imposent de repenser la manière de mettre en oeuvre la sécurité qui, encore aujourd'hui, laisse ouvertes des problématiques de vulnérabilité mémoire. De plus, les risques d'exploitation de ces vulnérabilités grandissent avec l'émergence de nouveaux cas d'utilisation (environnements intelligents de manière générale) impliquant des systèmes de plus en plus complexes et l'explosion du nombre de systèmes connectés (notamment pour des besoins de mise à jour à distance). En conséquence, des cyber-attaquants peuvent tirer profit de telles vulnérabilités pour prendre le contrôle à distance de ces systèmes connectés de façon massive.

Dans ce cadre, la thèse propose d'élaborer un noyau destiné aux petits objets qui soit capable d'apporter des garanties fortes d'isolation mémoire. Elle étudie l'association entre flexibilité élevée et forte sécurité au sein d'objets contraints pour une sécurité dès la conception sans perte fonctionnelle. Elle est constituée de deux contributions principales qui répondent aux attaques logicielles sur la mémoire.

La première contribution est un noyau de système d'exploitation, appelé Pip-MPU, qui propose une solution d'isolation ancrée dans le matériel et offrant une flexibilité dépassant les solutions actuelles. Pip-MPU est adapté du protonoyau Pip initialement destiné à des ordinateurs généralistes dotés d'une plateforme matérielle plus fournie et différente de celle des objets contraints. Pour cela, le noyau conçu est une refonte totale de Pip et propose un mécanisme de sécurité basé sur la *Memory Protection Unit (MPU)*, une unité du processeur, qui permet un contrôle d'accès matériel aux ressources. Malgré les fortes limitations imposées par la plateforme matérielle, Pip-MPU est aussi flexible que ce que permet la MMU en termes de sécurité. Du haut de ses 10 Ko de code et 16% de surcoût en termes de performances et de consommation d'énergie, Pip-MPU réduit le nombre d'opérations privilégiées exécutées de 99% et la surface d'attaque de la mémoire accessible depuis l'application de 98%.

La deuxième contribution est l'obtention de garanties fortes de l'isolation par l'usage de méthodes formelles. Plusieurs services du noyau ont été formellement prouvés pour l'isolation à l'aide de l'assistant de preuve Coq. Les propriétés prouvées sont les propriétés de sécurité de Pip imposant son modèle de sécurité d'isolation stricte. A notre connaissance, aucun autre système de l'état de l'art proposant de l'isolation par MPU n'a été formellement prouvé auparavant. Nous développons des nouvelles techniques de conduite de preuve et proposons de nouvelles métriques permettant de suivre l'effort de preuve et d'évaluer les hypothèses soutenant les preuves.

Toutes les contributions de la thèse sont en source ouverte.

Mots clés : isolation mémoire, objets contraints, mpu, vérification formelle, coq, pip, sécurisé dès la conception