Formal models and verification of memory management in a hypervisor

Pauline Bolignano

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Pauline Bolignano

préparée à l'unité de recherche 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires

Formal Models and Verification of Memory Management in a Hypervisor

Thèse soutenue à Rennes
le 24 mai 2017

devant le jury composé de :

Mads DAM
Professeur, KTH Royal Technical University / Rapporteur

Marie-Laure POTET
Professeure, Ensimag / Rapporteuse

Delphine DEMANGE
Maître de Conférence, Université de Rennes 1 / Examinatrice

Mario SUDHOLT
Professeur, Institut Mines Télécom / Examinateur

Thomas JENSEN
Directeur de Recherche, Inria / Directeur de thèse

Vincent SILES
Ingénieur Docteur, Prove & Run / Co-directeur de thèse

Contents

List of Figures

Remerciements

Tout d'abord un grand merci à Thomas Jensen et Vincent Siles, qui m'ont accompagnée tout au long de ce travail de thèse. Merci pour votre enthousiasme, votre optimisme et tout ce que vous m'avez appris.

Merci à Marie-Laure Potet et Mads Dam de m'avoir fait l'honneur de rapporter ma thèse. Merci pour vos retours et remarques judicieuses et constructives. Je remercie Mario Sudholt et Delphine Demange d'avoir bien voulu faire parti de mon jury, ainsi que pour leurs retours sur le manuscrit.

Cette thèse ne serait pas possible sans son principal sujet d'étude, l'hyperviseur développé par l'équipe SecT de TU Berlin. Je remercie le Professeur Jean-Pierre Seifert de m'avoir accueillie dans son équipe, ainsi que tous les membres de son équipe qui ont pu répondre à mes nombreuses questions. Je remercie tout particulièrement Michael Peter, pour son temps et sa gentillesse.

Je remercie l'EIT Digital pour leur financement lors de mes déplacements à l'étranger, à Berlin puis durant mon stage de trois mois à Dresde. Je remercie Prove & Run ainsi que mes directeurs de thèse sans qui ces déplacements n'auraient pas été possible.

Durant ma thèse, j'ai passé 4 mois à l'Inria Rennes, dans l'équipe de Thomas Jensen. Un grand merci à l'équipe Celtique pour son accueil chaleureux, et en particulier à tous mes collègues doctorants.

Je voudrais remercier tous mes collègues de Prove & Run. Merci tout particulièrement à Olivier, Stéphane et Benoit, pour leur disponibilité et leurs explications qui m'ont beaucoup aidés dans mon travail. Cela a été très stimulant pour moi d'être dans un environnement si riche.

Je remercie mes amis Aurélie, Camille, Florian, Juliette, Marie, Marion, Raphaël ainsi que mes soeurs, Clarisse et Lucie, d'avoir fait le déplacement pour assister à ma soutenance. Cela m'a fait très plaisir.

Un très très grand merci à mes parents, pour leur présence, leur soutien et leur douceur durant ces trois années de thèse (et les 25 précédentes). Un merci tout particulier à mon père pour avoir su rendre contagieuse sa passion pour son travail.

Enfin, je remercie Paul. Merci pour ton soutien, mais surtout merci pour ta curiosité et l'intérêt que tu as porté à mon travail tout au long de ma thèse.

Résumé en Francais

Les nombreux bugs et attaques découverts ces dernières années montrent que la plupart des technologies et des services que nous utilisons (téléphone, montre, ordinateur, voiture connectée, boites mail, applications) sont peu sécurisés. Citons par exemple l'attaque sur les voitures connectées par laquelle une tierce personne pouvait prendre le contrôle d'une voiture à distance. L'attaque des comptes Yahoo, découverte en 2016, qui a touché plus d'un milliard d'utilisateurs. Ou encore le bug très médiatisé Heartbleed de OpenSSL, qui permettait à un attaquant de lire des parties de la mémoire du serveur et du client, pouvant ainsi récupérer les clés privées de ces derniers.

Or les systèmes électroniques sont utilisés par un nombre toujours plus important d'utilisateurs. A titre d'exemple, plus de deux milliards de personnes dans le monde possédaient un smartphone en 2016, et ce chiffre pourrait bien s'élever à six milliards d'ici 2020. Parallèlement à cela, la sensibilité des données que manipulent ces technologies, la criticité de ce qu'elles contrôlent, et leur omniprésence dans nos vies amplifient grandement l'impact d'un bug. La sécurité devient donc une préoccupation majeure. On peut alors se demander comment augmenter le niveau de sécurité d'un logiciel.

Les programmes dont les failles de sécurité peuvent s'avérer critiques sont testés de manière intensive. Le test est une étape incontournable du développement logiciel, il permet de trouver les bugs au cours du développement, et de s'assurer que, pour certaines entrées, le programme réponde conformément à la spécification. Cependant, si le test permet de montrer l'existence de bugs, il ne permet pas d'en prouver l'absence. En effet, les tests ne peuvent pas couvrir tous les cas d'exécution possibles. De plus, sans même parler de couverture, certaines propriétés du programme sont difficilement vérifiables par des tests. Par exemple, la notion de confidentialité est difficile à appréhender par le test car la lecture d'une donnée n'a pas d'effet visible sur le système.

La preuve formelle de programme permet non seulement de considérer *tous* les cas d'exécution possible, mais aussi d'exprimer des propriétés de haut niveau, comme la confidentialité. C'est un processus long et coûteux. Pour ces raisons, elle est encore peu pratiquée dans l'industrie. Cependant, le coût et les dégâts que peut générer un bug rendent l'utilisation des méthodes formelles incontournable dans certains domaines. De plus, la maturité des outils permet maintenant de prouver formellement des systèmes larges et complexes, comme les systèmes d'exploitation (SE) [Les15; Kle09].

Ce sont justement les SE qui sont au cœur de notre étude. Le SE est la première couche de logiciel qui s'installe sur le matériel. C'est lui qui le paramètre, et qui gère l'accès aux ressources. Il est important qu'il soit sécurisé, car il peut compromettre la sécurité de tous les programmes qui tournent au-dessus de lui. Plus exactement, c'est la base de confiance du SE que l'on doit vérifier. C'est-à-dire la plus petite base de code sur laquelle un bug peut mettre en péril le système en entier au regard d'une propriété. Il convient donc de réduire cette base de confiance afin de réduire les risques de bugs, et faciliter la preuve. Sur les SE monolithiques, tout le système tourne en mode privilégié, donc tout le système a accès aux ressources. Dès lors, il est difficile de réduire la base de confiance. Les SE à micro-noyau, en revanche, ne font tourner en mode privilégié uniquement le nécessaire (le noyau correspond à la partie du SE qui tourne en mode privilégié, d'où le

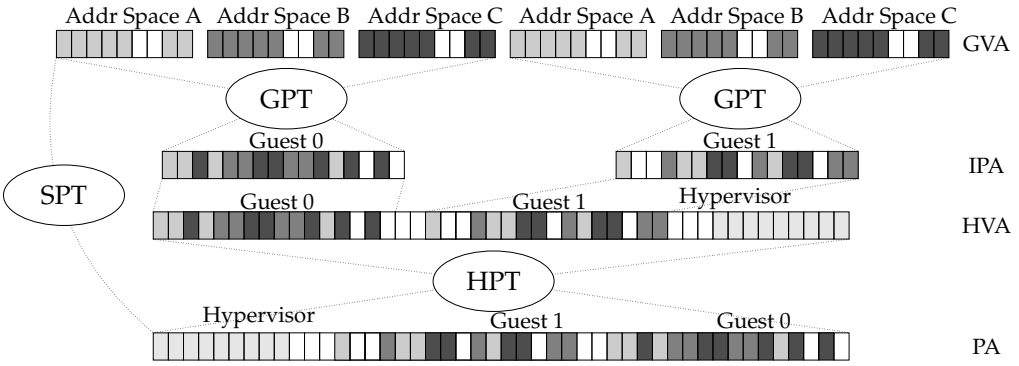


FIGURE 1: Shadow Page Tables

nom). Il est donc possible, avec ce genre d'architecture, de réduire la base de confiance. Ainsi ces systèmes sont de meilleures cibles pour la preuve que les précédents.

Notre travail porte sur un hyperviseur à micro-noyau, à base de confiance réduite. Un hyperviseur est un SE particulier, sur lequel plusieurs SE peuvent eux même tourner. L'hyperviseur *virtualise* donc les ressources pour les SE invités, qui eux même les virtualisent pour leur processus. Nous nous sommes intéressés à des propriétés d'*isolation* de la mémoire des SE invités. L'isolation peut être découpée en deux parties, l'intégrité et la confidentialité, que l'on définit comme suit:

- L'intégrité assure que la mémoire d'un SE invité ne peut pas être altérée par un autre SE invité, à part lorsqu'il a expressément donné la permission de le faire.
- La confidentialité assure que la mémoire d'un SE invité ne peut être lue par un autre SE invité, à part lorsqu'il a expressément donné la permission de le faire.

L'accès à la mémoire est virtualisée par le SE par le biais des *tables de pages*. Le SE maintient des tables de traduction, qui traduisent des adresses virtuelles en adresses physiques. Plus exactement, elles traduisent des *pages* d'adresses virtuelles vers des *pages* d'adresses physiques pour des questions de performance, d'où le nom *table de pages*. Le SE indique au matériel quelle table de pages utiliser pour effectuer la traduction. Un processus manipule des adresses virtuelles, qui sont traduites à son insu par le matériel en adresses physiques, en utilisant la table indiquée: la mémoire est *virtualisée*. Si une adresse physique n'est pas référencée par une adresse virtuelle dans la table de pages, elle n'est pas accessible. C'est donc bien le SE qui gère l'accès à la mémoire physique.

L'hyperviseur ajoute encore une couche de virtualisation. Nous présentons la solution de virtualisation par Shadow Page Tables, qui est utilisée dans l'hyperviseur sur lequel nous travaillons. Comme le montre la Figure 1, le SE invité gère des tables de pages pour virtualiser la mémoire de ses processus (Guest Page Tables, GPT). L'hyperviseur gère lui même des tables de pages pour virtualiser la mémoire de ses invités (Host Page Tables, HPT). Lorsque un invité tourne, il manipule des adresses virtuelles qui, pour être traduites en adresses physiques, devraient êtres traduites successivement par les tables de pages de l'invité puis par celles de l'hyperviseur. Or, sur certaines architectures, nous ne pouvons indiquer au matériel qu'une seule et unique table de pages pour la traduction d'adresses. L'hyperviseur crée donc, pour chaque invité, une table de pages qui combine les tables de pages de l'invité et les siennes, que l'on appelle les Shadow Page Tables (SPT).

Notre étude s'intéresse principalement à la vérification de l'algorithme de SPT : nous vérifions qu'il assure bien l'isolation de la mémoire des invités. Nous utilisons pour cela

une méthode de *raffinement* (ou *abstraction*). Le principe d'une telle méthode est de montrer que notre système concret correspond à un modèle abstrait, idéalisé, pour pouvoir prouver les propriétés de haut niveau sur ce modèle abstrait. Pourvu que l'abstraction respecte certaines contraintes, les propriétés prouvées au niveau abstrait sont valables au niveau concret. L'intérêt d'une telle méthode est qu'elle permet d'exprimer les propriétés dans un formalisme de haut niveau, et de raisonner sur un modèle plus simple. En effet, dans un modèle de code concret, beaucoup d'opérations et de structures sont complexifiées pour des questions d'optimisation et de conformité avec le matériel, c'est le cas par exemple des tables de pages. Sur un modèle si complexe, il devient difficile de prouver des propriétés telles que l'isolation, ou même seulement de les exprimer. En fait, la clarté avec laquelle est formulée la propriété est une composante importante de la confiance: si la propriété est elle même trop complexe pour pouvoir être comprise facilement, alors il existe un risque que cette propriété n'aie pas le sens voulu.

Notre contribution est double. La première est la preuve, à un niveau de confiance élevé, que l'algorithme de gestion de la mémoire assure l'isolation des SE invités. Nous expliquons le concept de haut niveau de confiance à la Section 2.5, il tient en partie au fait que notre niveau abstrait est très épuré, la notion de table de pages y a en effet été totalement abstraite.

Notre seconde contribution est méthodologique. Dans la littérature, il existe de nombreux livres sur les meilleurs pratiques de développement logiciel. Ils sont le résultat de plusieurs années de retour d'expérience, de la part des académiques et des industriels. La preuve formelle de logiciel n'est pas aussi répandue que le développement logiciel ou le test, et ce particulièrement dans l'industrie. Dans cette thèse, nous présentons précisément notre méthodologie. Nous montrons, entre autres, les propriétés du modèle concret et l'interdépendance de leur preuve ainsi que la manière de concevoir le modèle abstrait.

Le Chapitre 1 présente le contexte et les principales notions qui permettent de comprendre notre travail et sa portée. En particulier, nous revenons sur les concepts d'hyperviseur, de sécurité et de preuve formelle. De plus, nous présentons succinctement le langage et l'assistant de preuve développé par Prove & Run, que nous utilisons pour nos modélisations et preuves.

Dans le Chapitre 2, nous faisons un état de l'art de la preuve formelle d'hyperviseurs et de SE. Nous expliquons plus en détails le principe de preuve par abstraction, et comparons notre application de cette méthodologie par rapport à celle faite dans d'autres projets.

Nous détaillons le modèle concret de l'hyperviseur ainsi que toutes les transitions du système dans le Chapitre 3. De plus, nous présentons notre modélisation des tables de pages.

Nous présentons les invariants de notre système dans le Chapitre 4. Les invariants sont des propriétés valables dans tous les états du système. Nous établissons des propriétés sur les transitions du système, et nous spécifions et prouvons les effets des transitions liées à la mémoire, et de celles effectuées par les SE invités. Nous prouvons la préservation des invariants sur ces mêmes transitions.

Ces invariants et propriétés du système concret sont essentiels pour prouver la correspondance avec le modèle abstrait, que nous présentons dans le Chapitre 5. C'est dans ce chapitre que nous présentons également la fonction d'abstraction, les propriétés de sécurité et leur preuve ainsi que les preuves de raffinement.

Introduction

The numerous bugs and attacks discovered over the past years show that the technologies that we use in our daily life are not as secure as we would want them to be. We can cite the attack against the entertainment system of a smart car, through which an attacker could gain control of the driving system. The attack on the Yahoo accounts, discovered in 2016, has impacted more than one billion users. Finally, the famous Heartbleed attack over OpenSSL allowed an attacker to retrieve the private keys of client and servers, rendering the communication insecure. The fact that technologies handle sensitive data and control critical mechanisms, and that their use is pervasive in our lives increase the negative impact of a bug. There is therefore a high need for security.

The first step toward security is testing. Tests indeed allow to uncover early bugs and verify that, for a certain set of inputs, the program behave as intended. However, if tests allow to find bugs, they cannot prove their absence. The use of formal methods, on the contrary, allow to reason about *all* the possible execution paths of a program. It is usually more costly and time consuming than testing, yet it is the only mean of ensuring that a system is compliant with its specifications.

We are interested by operating systems (OSes). An OS is a peace of software that runs directly on the hardware. It manages the hardware resources for the processes, and controls the access to them. More precisely, we study a hypervisor, which is a particular kind of OS that runs several *guest* OSes on top of itself. Just as an OS does for processes, an hypervisor manages and virtualize resources for the guest OSes. It is therefore an important target for security, as a bug in the hypervisor might compromise all the systems running on top of it.

In this thesis, we present a formal proof that the memory management in a hypervisor provides memory isolation of the guests. We proceed by abstraction, meaning that we design an abstract model of the hypervisor and prove its correspondence with our concrete model. The properties of isolation are proved on the abstract model and transferred down to the concrete model.

We present the main concepts in Chapter 1. Among others, we present the management of memory in hypervisors, we introduce the notion of security, we give an overview of the existing tools for formal methods and present the one we use.

In the Chapter 2, we review the state of the art of formal proofs about OSes and hypervisors. We explain the principle of proof by refinement, and we compare our application of this methodology to the one done in other projects.

We detail the concrete model of the hypervisor, along with its transitions in Chapter 3. Furthermore, we present our modeling of the PTs.

We present the invariant properties of our concrete model in Chapter 4. We also specify and prove the effects of the transitions related to the memory management and to the guest execution, and prove the preservation of invariants over these transitions.

These properties we prove are essential for the proof of correspondence between the concrete and abstract model, that we present in Chapter 5. We also present in this chapter the abstraction function, the security properties and their proof, and the refinement proofs.

Chapter 1

Context

Contents

This thesis evolves around three concepts: *hypervisors*, *security*, and *formal methods*, which we introduce in this chapter.

A *hypervisor* is a particular kind of Operating System (OS). An Operating System (OS) is a layer of software that manages the resources of the hardware. All the applications run on top of it. A hypervisor is an OS which can run other OSes on top of itself. We introduce OSes and hypervisors in Section 1.1.

In order to build secure systems, the base of the system, that is the OS, must be secure. Indeed, using secure applications might be worthless if the security mechanisms can be bypassed at the OS level. This is why we are concerned with their security. Yet, what does *security* means?

The term *security* highly depends on the system we are considering, and of its use. For example, in a car system, one must prevent an action of the entertainment system to modify the driving system part. One may also define which information flow are allowed or not. For example, in modern cars, the volume of the audio is increased when the speed goes up, in order to mask the noise of the engine, meaning that a flow from the driving system to the entertainment system is authorized. Similarly, data provided by the GPS are passed to the entertainment system, but it should not be leaked to a third party. We present the main kinds of security properties in Section 1.2.

We then introduce the notion of formal methods in Section 1.3. They allow to formally ensure that the design and the implementation of a system enforce the targeted

security properties. The formal approach is the most trustworthy approach, and is required to reach the highest level of some software certifications. We will briefly develop certification issues in Section 1.4.

1.1 Hypervisors

1.1.1 Operating System Kernels

An OS is a layer of software that interfaces directly with the hardware through a defined set of instructions implemented by the hardware. It provides an abstraction of the hardware to processes running on top of it. Consequently an OS decides which part of the hardware resources are exposed to the processes, thus having the complete control over the resources access. Another consequence is that the implementation of a user program can usually be independent of the underlying platform (e.g. ARM [Arm] or Intel [X86]).

More generally, an OS virtualizes resources, i.e. it does not only abstract them but also multiplex them. Virtualization gives processes the illusion of running alone on the OS, whereas the resources are shared by several processes. Virtualization allows the OS to share the resources, but it also contributes to securely isolate resources of the different processes. We will develop the virtualization of memory in more depths in Section 1.1.3.

Kernels Hardware architectures provide at least two modes, a unprivileged mode and a privileged mode. The code running in privileged mode has access to all the resources, while only a restricted set of resources is available to code running in unprivileged mode. The OS, more exactly the **kernel** of the OS runs in the most privileged mode. The kernel of an OS is the mandatory part of the OS common to all other software. Common OSes such as Linux, OS-X or Windows have monolithic kernels, meaning that the kernel corresponds to the whole OS. It implies that all the modules such as the file system or the device drivers run in the privileged mode. Consequently, a bug in one module jeopardizes the whole system.

On the contrary, in a **micro-kernel** based OS, everything that can be put outside the kernel is removed from the kernel [Lie95]. Micro-kernels are more modular, as depicted in Figures 1.1. They are therefore intrinsically more robust and are particularly convenient to ensure a good separation between components.

The kernel is always part of the base of code which is critical for the system security, called the Trusted Computing Base (TCB). Since micro-kernel based OSes have smaller kernels, they usually have a reduced TCB. Yet some components can hardly be removed from the TCB, or even from the micro-kernel. The memory virtualization mechanism for example controls the accesses to memory. If not trusted, the memory can be corrupted or leaked. Similarly, the basic IPC mechanism, which manages the message passing operation between processes, should be trusted. The process manager, which decides where to allocate new processes, is sometimes implemented outside the kernel, but performs too sensitive operations to be put outside of the TCB [Les15]. The scheduler, which decides which process is to be run, is considered as sensitive for some particular applications, as it can prevent a process to run.

1.1.2 Different Types of Hypervisors

A hypervisor is a particular kind of OS that introduces an additional level of virtualization. Hypervisors virtualize the whole hardware, so that several OSes run on the same

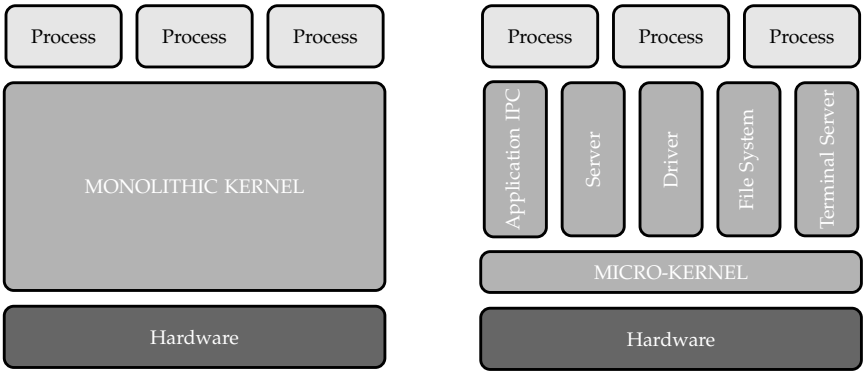


FIGURE 1.1: Monolithic versus Micro-Kernel based OS

platform. This way, the resources virtualized by OSes for processes are themselves virtualized by the hypervisor. We call an OS running on top of a hypervisor a *guest* OS, or guest.

The hypervisor provides a *virtual machine* to the guests running on top of it. The definition given by Popek and Goldberg in 1974 of a virtual machine is the following: "A virtual machine is taken to be an efficient, isolated duplicate of the real machine" [PG74]. It is also specified in [PG74] that hypervisors should:

- Provide an environment almost identical to the original machine.
- Not affect performances too much.
- Be in complete control of system resources.

A hypervisor might run directly on the hardware, we call it *bare-metal* or *type-1* hypervisor. It is the case for Xen [Xen] or VMware ESXi [Esx]. Or a hypervisor might run on top of an OS, as an application, in this case we call it a *type-2* hypervisor. VMware Workstation [Ws], VirtualBox [Vbx] or QEMU [Qem] are type-2 hypervisors.

A guest OS expects to run directly on the hardware, in a privileged mode. Yet when it runs on a hypervisor, the hypervisor runs on the most privileged mode, thus limiting the capacity of the guest. The hypervisor needs to be able to spot and virtualize all the instructions. Usually, if the guest makes a privileged instruction, the hardware switches to the most privileged mode, the hypervisor virtualizes the instruction and restores the execution of the guest. However, some guest instructions are non-virtualizable. They would just fail silently, thus preventing the hypervisor to be aware of it and virtualize them. For instance, in ARMv7, access to privileged bits of the current program status register (CPSR) has an undefined behavior in unprivileged mode [DN10]. Three main solutions exist:

Binary Translation The first solution, used by VMware since 1998, is the binary translation [Vir]. Basically, the hypervisor analyses the binary of the guest instructions and redirect them on the fly to virtualized instructions. This solution incurs overhead. However it does not require the guest to be modified, the guest is said to be *fully-virtualized*. It is thus highly compatible with any legacy OS.

Para-virtualization The second solution is called para-virtualization (or OS assisted virtualization). This solution implies that the guest is aware of virtualization. The hypervisor provides an interface to the guest, so that the guest can call it through hypercalls to perform the instructions on its behalf instead of trying to access directly privileged instructions. As the guest is modified, the compatibility might be poor, depending on how much the OS is modified. On the other hand, modifying the guest is a chance to optimize the calls and enhance performances. Para-virtualization has proved to be an efficient solution [Chi07].

Hardware Assisted Solution This solution was introduced ten years ago on Intel and AMD platforms, through virtualization extensions (Intel VT and AMD-V). It was later introduced on ARM. Hardware virtualization extensions provide additional registers and levels of privileges. The guest can run in a level of privilege higher than user level but lower than hypervisor level. Privileged registers are duplicated so that a guest can modify registers without trapping to the hypervisor. Consequently, the hypervisor does not need to virtualize each privileged instruction, it only has to parameterize the hardware virtualization extensions. Hardware virtualization solution allows to run fully virtualized guests, without needing to provide a binary translation. This solution is thus the simplest and the most portable of the three solutions presented.

We work on a para-virtualized, type-1 hypervisor. In the next section we introduce details about the Memory Management Unit (MMU). We present in more depths the hardware assisted and para-virtualization solutions for memory virtualization.

1.1.3 Memory Virtualization

When memory is virtualized, each entity runs as if it had the whole memory for itself, while the underlying platform shares the memory between several entities. In a classic OS with MMU, the OS manages the translations from virtual to physical addresses. In the case of hypervision, a level of translation is added. The hypervisor may either use a hardware virtualization extension (if available) or implement a virtualization mechanism in software.

Memory Management in an OS

A classical OS maintains some tables of translation from virtual to physical addresses [SGG12, Chapter 9]. This tables are called Page Tables (PTs), basically because the memory is decomposed into *pages*, and memory mappings between virtual and physical addresses are done at the granularity of a page. The hardware MMU uses the PTs maintained by the OS in order to translate virtual addresses to physical addresses. More specifically, the OS indicates to the hardware which PT to use through a register of the MMU (CR3 for x86, TTBR0 for ARM).

We illustrate a two levels PT in Figure 1.2. The virtual address of a page is decomposed into two indexes. The first is an index in the first level PT whereas the second index is an index in the second level PT. The entry fetched in the first level PT contains the address of a second level PT. We take the entry corresponding to the second level index in this second level PT. This entry contains the address of a physical page, with rights associated to it. We will explain in more details the mechanism of translation in Section 3.2. As can be seen, many entries in the PTs are empty, because a mapping from a virtual to a

physical page is added only when needed. For this reason the use of several levels of PTs saves memory space.

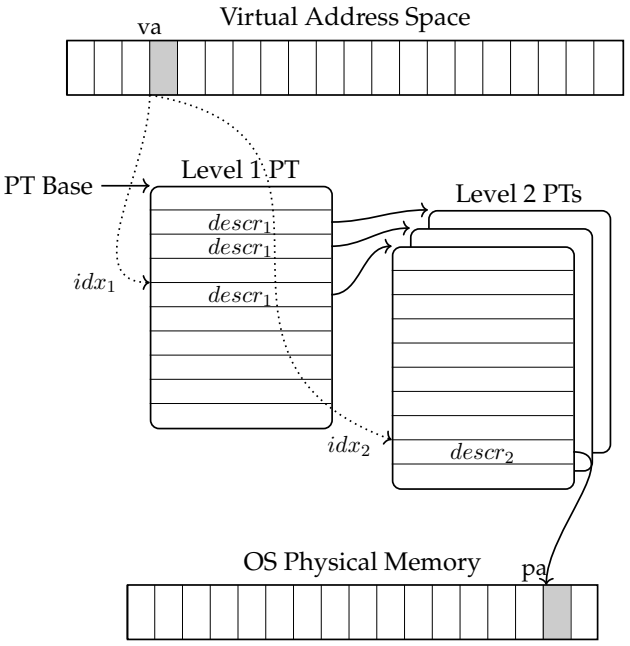


FIGURE 1.2: Two-Level Page Table

The OS maintains distinct PTs for each process and for itself. Each time a context switch is performed, the OS updates the dedicated MMU register to change the PT to be used by the MMU. The MMU ensures that if no mapping exists between a virtual and a physical address in the current PT, then this physical address cannot be accessed. If a process tries to access a virtual address which is not mapped in the current PT, then the MMU triggers a page fault. The OS then handles the fault. For example it may allocate a new physical page for the process and maps the virtual address to it in the PT, or just crash. The use of the MMU allows the OS to have the full control over the software accesses to the physical addresses, and thus helps ensuring isolation between processes.

Memory Management in a Hypervisor

In the case of a hypervisor, guest OSes still manage their own Guest Page Tables (GPTs). However, GPTs do not translate Guest Virtual Addresses (GVAs) to Physical Addresses (PAs) directly, as only the hypervisor has enough privileges to access PAs.

As depicted in figure 1.3, GPTs translate GVAs to Intermediate Physical Addresses (IPAs) which are not physical addresses, they correspond, up to a translation, to Hypervisor Virtual Addresses (HVAs). It means that they are virtualized by the hypervisor: another PT must be used to translate them into physical addresses. The Hypervisor Page Tables (HPTs) handle this second translation, from HVAs into PAs.

Hardware Assisted Solution With the hardware virtualization extensions [Smm], the MMU takes two PT pointers. Therefore the guest OS can specify which GPT it uses while the hypervisor specifies which HPT it uses. The MMU then handles the whole translation from GVA to PA. The virtualization being offloaded to the hardware, this solution is the most simple to implement and presents the highest performances.

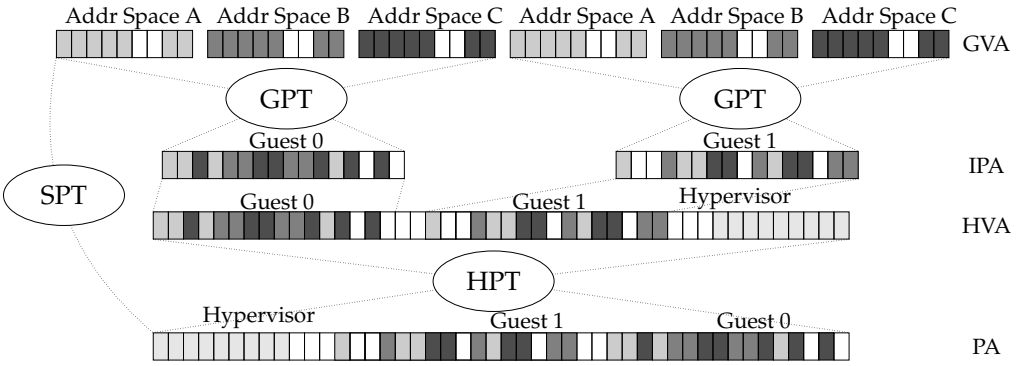


FIGURE 1.3: Shadow Page Tables

Para-virtualization: Shadow Page Tables The most common *software* solution is the one based on Shadow Page Tables (SPTs). SPTs are maintained by the hypervisor and translate the virtual addresses of the guest (GVA) to physical addresses (PA), as illustrated in Figure 1.3. For clarity's sake, we here consider that the IPA and HVA depicted in the Figure 1.3 are equal and refer to them as IPA. The hypervisor creates and manages the mappings of the SPT by combining the PTs of the guest (GPT) and the PTs of the hypervisor (HPT). For example when a page fault occurs at GVA gva, the hypervisor is notified. It goes through the GPTs to find out if any mapping from gva to a IPA ipa is present in the GPTs. If there is one, it computes the physical address pa corresponding to ipa and, provided the guest is allowed to access this part of the memory, it adds the mapping from gva to pa in the SPTs. If the gva is not present in the GPTs, a page fault is triggered by the MMU. The way of handling the page fault depends on the hypervisor considered. It can for example inject the page fault into the guest, so that the guest can add the mapping to the GPTs. Then the execution faults again on gva, because it is not yet in the SPTs, and it brings us back to the first case.

In order to keep the SPTs in synchronization with the GPTs of the guest, the hypervisor traps and emulates the Translation Lookaside Buffer (TLB) instructions performed by the guest. The TLB is a hardware cache that stores the most used address translations [ADAD14, Chapter 19]. It speeds up the translation because the hardware does not need to walk the PTs each time. When an OS modifies its PTs, it needs to keep the TLB in synchronization with the PTs. Basically the OS can invalidate a single entry or all the entries of the TLB. The SPT algorithm of a hypervisor intercepts these TLB instructions and emulate them, by removing one or several mappings from the SPT. That is why shadowing the PTs is often referred to as virtualizing the TLB.

Para-virtualization: Direct Paging Another way of handling translations when using para-virtualization is direct paging. In this case, the guest maintains direct translations from GVAs to PAs. It means that the IPAs depicted in Figure 1.3 are equal to the PAs, thus the hypervisor does not maintain any PT for the guest. In particular, there is no concept of SPT. The guest has only read access to the GPTs, thus every attempt of modification by the guest traps, so that the hypervisor retains control over the memory accesses.

This solution implies more modifications to the guests, so it is the less portable solution of the three presented. On the other hand, it simplifies the virtualization code, thus the implementation is less prone to error than for the SPT. Xen, which is one of the most wildly used hypervisor, uses direct paging [Chi07, Chapter 9].

1.2 Security Properties

The security property we target is isolation of the guests, more specifically, isolation of their memory and registers, as defined in [Les15]. Basically, we want to show that no guest is able to read or write to some predefined secret memory parts or registers of other guests. We do not prove functional properties on our system, for example, we do not ensure formally that the execution is never aborted. From a security perspective, the important point is that the security properties hold for every state of the system.

Our property of interest is a weaker version of non-interference. Indeed, non-interference is usually too strong for kernels, as we do want guests to interfere. We present non-interference in the next section, and weaker versions in Section 1.2.2.

1.2.1 Non-Interference

A non-interference policy defines which domain is allowed to flow information to other domains. We note $A \rightsquigarrow B$ if a flow is authorized from domain A to domain B. Given such a policy, the property of non-interference ensures that no other flow except the ones specified in the policy is allowed. Formally, the non-interference has been specified with a *purge* function [Rus92]. Consider an initial state of a system, and a sequence of actions from this state. We purge the actions not related to a domain A, i.e. we remove all the actions that are not allowed to interfere with domain A. We compare the output of the first sequence of actions, and the output of the purged sequence. If the two outputs are equals, then there is no flow except the one allowed by the policy. Intuitively, it means that A cannot learn anything about the previous sequence of actions, except from the action whose domain has a flow to A. Instead of being stated with the whole trace of events, non-interference can equivalently be stated with the unwinding condition, which is a step-wise property [Rus92]. Usually this condition is used for mechanized proofs instead of the purge version, for its convenience.

Non-interference is strong, and does not make sense for processes (resp. guests) when they are allowed to communicate because they do interfere. Usually we consider weaker properties derived from non-interference.

1.2.2 Variants of Non-Interference

Intransitive Non-Interference In a non-interference policy, if $A \rightsquigarrow K$ and $K \rightsquigarrow B$, then $A \rightsquigarrow B$ must be authorized. As the name suggests, intransitive non-interference is not transitive. It means that A and B are considered non-interferents as long as the flow of information is passed through an authorized channel, in our example the channel is K. Intuitively, in an OS, K would be a part of the kernel, and A and B two processes. This property can also be stated with a purge function or an unwinding condition.

We will introduce later seL4 [Kle+09], a formally proved micro-kernel. Klein et al proved a more general version of intransitive non-interference. Basically in their definition, they use a weaker unwinding condition than the one of intransitive non-interference, and the domain of an action is not defined only by the action itself but also by some parts of the current state.

Isolation The property that we target is isolation of the guest OSes, which is also a weaker version of non interference [Les15]. Isolation of the guest OSes can be decomposed in two sub-properties: *integrity* and *confidentiality*. The integrity property for one

guest ensures that its resources are not modified by other guests, unless it has given the authorization to do so. The confidentiality for one guest ensures that executions of other guests do not depend on its resources, unless it has given the authorization to do so.

Integrity is easier to state than confidentiality. Indeed the effects of an integrity flaw are easily observable on a trace of execution: a data has been modified. Confidentiality is not directly observable, it implies to compare two traces of executions. We express confidentiality in a similar fashion as what is done with the purge function for non-interference. Let g be a guest, s and t two states. We write $s \underset{g}{\sim} t$ if s and t are equal except on the secrets of guest g. We compare two finite traces of execution from s and t, in which g does not run. Then we verify that the resulting two states s' and t' verify $s' \underset{g}{\sim} t'$. Intuitively, it means that the execution of other guests than g does not depend on the secrets of g, hence the other guests do not know about g's secrets. If a secret is shared by several guests, then we consider a trace of execution where none of the guests sharing the secret might have run.

The integrity property is similar to the *non-exfiltration* property presented by Nemati et al. in [NGD15]. Confidentiality is similar to their *non-infiltration*. We detail their work in Chapter 2.

1.3 Formal Methods

The first step to build a correct or secure program is to make tests. Tests allow to check the functionalities of a program, and help the programmer to detect bugs at early stage of development. However tests are not exhaustive. Among the infinity of possible execution path of a program, only a few are actually tested. The utility of a test suite is measured by its code coverage, i.e. the part of the code explored by the tests.

Some automated test tools, like fuzzers, allow to improve the code coverage. Fuzzers [Afl; Pea] execute a program continuously, with different inputs each time. They instrument the program and are able to chose inputs to extend the number of paths explored. They are widely used in industry, and they are also used by attackers to find and exploit flaws in the code.

Yet even advanced testing methods cannot cover all the possible execution paths. Formal methods do.

1.3.1 Tools for Theorem Proving

Formal method tools rely on mathematics to specify and verify programs. Contrarily to testing, which proves the *presence* of bugs, formal methods show the *absence* of bugs. We present below three types of tools, but our list is not exhaustive [Alm+11, Chapter 2].

Model Checkers Model Checkers are automated tools for verifying finite state program [Cbm]. They generate an abstraction of the program, a model, and verify that all the states of the model respect some specifications. The state space of the model is verified exhaustively. Model checkers are automated tools, they require no human interaction for proofs. However the execution time and memory consumption rise exponentially with the number of states. The properties can only be proved on small portion of the program, or on a reduced setup. Bounded model checkers reduce the state space by only unwinding loops a bounded number of times.

SMT Solvers Satisfiability Modulo Theories (SMT) solvers are able to decide whether a first order formula is satisfiable regarding a certain theory. The level of expressiveness of the properties is generally reduced. However these tools present the advantage of being fully automated. Among the most famous SMT solvers, we can cite Z3 [Z3], alt-Ergo [Alt] and Yices [Dut14].

Interactive Theorem Provers Interactive theorem prover such as Coq [Coq], Isabelle [Isa] or Prove & Run tools, are not fully automated, but they allow to prove more expressive properties. For example, Coq and Isabelle allow to express properties of higher order logic. Because the proofs require interaction with the user, the use of such tools is costly and less adaptable than the previous ones. On the other hand, only this kind of tools enables to write such expressive properties.

1.3.2 Methods for Theorem Proving

Different methods can be used to achieve formal verification of some properties on a system. We present two kinds of methods which have proved suitable for OS verification.

Annotations

Annotations are used to prove **behavioral properties** of a program. Pre and post conditions specified at various steps of the program generate proof obligations that are discharged automatically using some SMT solvers or model checkers, or interactively using interactive theorem provers.

Some tools, such as Dafny [Lei10], require a modeling of the program in a specific language. The annotation are then written in this same language.

Other tools allow to write annotations directly into the C code. The ANSI/ISO C Specification Langage (ACSL) is an annotation language for C code, part of the Frama-C project [Acs]. The Verifier for concurrent C code VCC [Coh+09] also allows to write annotations directly into the C code. VCC provides means to reason about concurrency. For example it is possible to specify by whom an element can be modified at each point of the program, thus making it possible to reason about shared values in a concurrent environment [Mos+09]. VCC has been used to achieve a large project of system verification, as we will show in Section 2.3.2.

Modeling and Interactive Proving

Other tools such as Coq, Isabelle, Why3, event-B are preferred when proving **properties on the whole model** [Coq; Isa; Why; Eve]. The models along with the properties can be written in the corresponding specification language. The proof goals are either verified with the integrated theorem prover (this is the case for Coq or Isabelle), or verified with external theorem provers (this is the case for Why3 and event-B). For example, Why3 is compatible with 16 SMT solvers, and 3 interactive provers.

We use a tool developed by Prove & Run, which belongs to the second category. The model and the properties are written in the same Specification and Modeling language, called Smart. C code can then be generated from Smart.

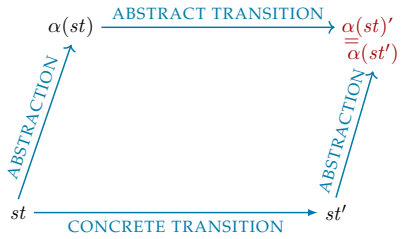


FIGURE 1.4: Commutation Diagram

1.3.3 Proof by Abstraction

When verifying properties, one rely on low-level properties, such as the absence of overflows or out of boundaries accesses. These low-level properties are needed in order to prove more elaborated ones. For example, as we will explain in more detail in Section 3.2, a PT is composed of several tables, each entry of a table leads to another table or to a physical page with some rights associated to it. If the goal is to specify which mapping from a virtual to a physical address is present in a PT, one would have to prove first the low-level properties mentioned previously, so that the PT can effectively be interpreted as a partial function from virtual to physical addresses with rights.

At some point, on large systems, it becomes difficult to focus on high-level properties without being overwhelmed by low-level details. Furthermore, high-level properties may be difficult to express, because they lack high-level structures.

The proof by refinement (or abstraction) addresses this problem. The principle is to build an abstract model, which uses abstract structures for which the low-level properties intrinsically hold. For example, in the work we present in this manuscript, we have totally abstracted the structure of PT. The abstract memory of a guest is composed of memory cells, which are tagged with some rights. These tags are what remains of PTs after abstraction. The link between the concrete and the abstract model is formally proved. To do so we prove that if an abstract state corresponds to a concrete state, and if a transition is possible from this concrete state, then an abstract transition is possible and the resulting abstract state corresponds to the resulting concrete state. This is also referred to as commutation proof, and is illustrated in Figure 1.4.

The interesting result of such an approach is when the properties proved on the abstract state also hold on the concrete state. This is not true for every property, we will develop this aspect in deeper details in Section 2.4. In our case, the key argument for preservation are that we use a function from concrete to abstract models, instead of a relation, and that both our models are deterministic.

The proof by refinement has been successfully applied to some large projects, such as the formally verified micro-kernel seL4 [Kle+09], or ProvenCore, the verified OS micro-kernel developed by Prove & Run [Les15].

1.3.4 Prove & Run Tools

We used the tool suite developed by Prove & Run to carry our modeling and proofs. We give a short introduction to the tools.

The language is called Smart, and is used both to write the program and its specifications. It's a pure and functional language and does not support higher order functions. Rich properties can be expressed, in particular quantifiers are supported.

The prover is interactive, but can also discharge trivial goals automatically. The figure 1.7 shows a screenshot of the environment. We explain below the basis of the language and the prover, on a small example.

In Smart, a predicate is given input parameters and returns output parameters, but it can also pass information through an other kind of output, called labels. The output parameters are recognizable by the "+" on their right. The label is a sort of exit flag, and can take whatever value we want. Most of the time a predicate raises the labels true or false, or only true. Exit labels are declared after parameters with the following syntax: `->[label1, label2, ...]`.

Figure 1.5 shows the signature of a predicate in Smart. As we will see on Section 3.2, we work with two levels of PTs. The first level PT is merely an array of *first level descriptors*. The predicate of Figure 1.5 gets a first level descriptor from memory. It takes as input a memory and a physical address `ppde`. If the size of a first level descriptor plus the address `ppde` exceeds the size of the memory, then the access is out of memory boundaries, and the predicate raises `oob`. If not, the function returns the first level descriptor located at `ppde`, and raises `true`.

```
public get_fst_level_descr_p(mem mem, addr ppde,
       fst_level_descr descr1+) ~> [true, oob]
/* If the first level descriptor at address [pde] exceeds the
 * memory boundaries, raises [>oob].
 * Else raises [>true] and returns the first level descriptor
 * located at physical address [ppde] in the memory [mem].
 */
program { ...some implementation... }
```

FIGURE 1.5: Example of the Signature of a Predicate in Smart

A lemma is a predicate which we have to prove always raises true. We show in Figure 1.6 a lemma which states under which conditions the predicate just presented cannot raise `oob`. The notation `p =>` means *if the predicate p raises the label true*. The question mark in front of a predicate is a label transformer, it redirects all the labels other than true to false. The lemma in Figure 1.6 is to be read as *if all the premises raise `true` then the last predicate raises `true`*.

More specifically, a first level PT is a table containing maximum `NUM_DIR0_ENTRIES` entries. This lemmas states that if we have some well-formedness properties verified on valid guests, and if the guest we consider is valid, then accessing an entry at an index lesser than `NUM_DIR0_ENTRIES` in the first level PT of that guest succeeds.

The screenshot of the environment in Figure 1.7 shows that our lemma generates a proof obligation (on the right part of the screen). Each node corresponds to a tactic we have used: instantiate some quantifiers, compose with some lemmas, and unfold a definition. A right click on a proof node shows the list of available interactive tactics. The nodes are colored in red when the proof has not been completed, green otherwise. For example, the green node on the bottom of the screenshot corresponds to a path of the proof that has been completed.

As you can see in the proof view, the proofs are made to be humanly readable, in order to ease the audit of the code. This is an important point when it comes to certification.

```
public lemma vtlb_d0_get_fst_level(mem mem, vcpus vcpus,
valids valids,vcpu_idx vcpu_idx, addr d0p, uint32 sn,
uint32 i0)
/* There is no out of boundaries access when fetching a first level
 *  descriptor of a valid SPT.
 */
program
// Variables declaration
{{ vcpu_t vcpu, addr ppde }} {
        // Well formedness properties on SPTs of valid guests
        pool_vtlb_region(vcpus, valids) =>
        spt_vv_pool_d0(vcpus, valids) =>
        // The guest considered is valid
        is_valid(valids, vcpu_addr) =>
        // Get the guest
        ? get(vcpus, vcpu_idx, vcpu+) =>
        // [d0p] is the physical address of a SPT of that guest
        reachable_dir0_sn(vcpu, d0p, sn) =>
        // [i0] is a valid index
        lt(i0, NUM_DIR0_ENTRIES) =>
        // Get the address of the [i0]th entry in the SPT
        get_pde_addr(d0p, i0, ppde+) =>
        // Get the descriptor at this address
        ? get_fst_level_descr_p(mem, ppde, _ );
}
```

FIGURE 1.6: Example of Lemma Written in Smart

1.4 Certification

The **Common Criteria (CC)** is an international standard at the basis of certification [Cc]. It defines several Evaluation Assurance Levels (EALs). A high EAL level means that the claimed security properties are enforced with a strong level of assurance. The lowest levels of certification only require informal analysis, whereas the highest levels require semi-formal or formal analysis. The highest level of assurance, namely EAL7 requires formal proofs. The EAL6 level, just below, is semi-formal. It basically means that the specifications should be structured and not ambiguous. In particular, EAL6 does not require machine-checked proofs, except for the security policy model, as we explain below.

The CC prescribes the use of several abstraction models. The highest and more abstract model specifies the security policy, and must be formal for the two highest EALs (EAL6 and EAL7). The next lower model is the functional specification of the system. It should be semi-formal for EAL6 and formal for EAL7. Similarly, the last and lowest model, which is the implementation design, should be specified semi-formally for EAL6 and formally for EAL7. A formal proof of correspondence between these levels is required for EAL7, whereas EAL6 requires a manual demonstration.

The **DO-178** is another standard, targeted to develop avionic systems [Do1]. The critical aspect of the software is rated from E (a failure has no effects) to A (a failure is catastrophic). Given this level, the software development must be conducted following some particular guidelines. For example the level E is not submitted to any constraints, the level D must be documented, whereas the level A must be extensively tested. The most recent version of this standard, the DO-178C, introduces the notion of formal methods

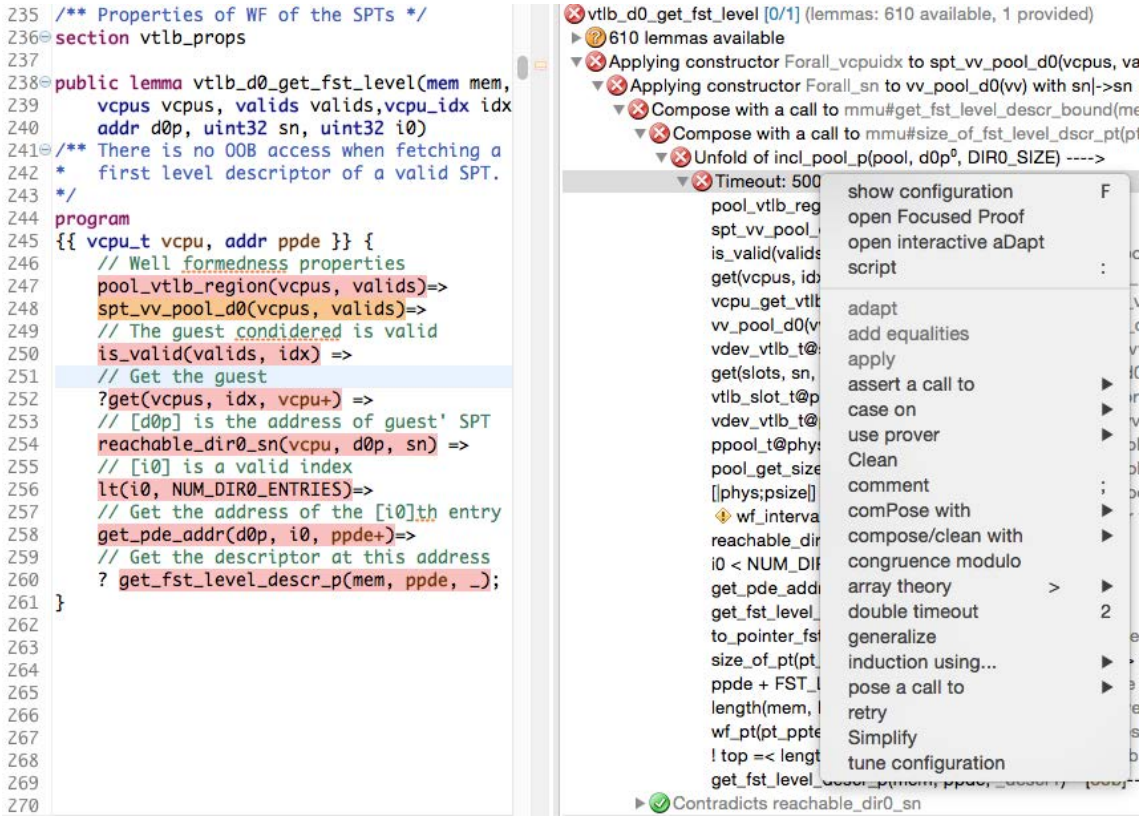


FIGURE 1.7: Screenshot of the Proof Environment

[Jac12].

The **ARINC 653** is a standard for separation kernel development for avionic application purposes. A system compliant with this standard is supposed to be able to run several processes with different DO-178 levels on the same platform in a safe manner. In other words it should be able to provide a certain degree of isolation. Zhao et al. have conducted proofs on a formalized model of the ARINC 653 standard in order to verify the information flow security properties [Zha+16]. They have proved that the standard presented some flaws, and have exhibited the presence of these covert channels in two ARINC 653 compliant separation kernels.

1.5 Key Points

- An hypervisor manages and virtualizes hardware resources for the guest OSes. The control of memory accesses is done through the management of PTs.
- We target the *isolation* of guests memory. The property relies on the good management of the PTs maintained by the hypervisor for the guests, called the *Shadow Page Tables*.
- We use the *proof by abstraction* technique. We show the correspondence between a low-level and a high-level model, where the properties holds intrinsically.
- We use Prove & Run tools to write our models, express our properties and prove them.

Chapter 2

State of the Art

Contents

Forty years ago already, OSes were recognized as being a crucial component for security. Some early verification projects aimed at verifying properties of OSes. However tools for formal methods were not mature enough, and did not allow to consider realistic OSes, nor to conduct the proof in reasonable time. We outline these projects in Section 2.1.

The system verification field grew in importance in the years 2000. Several huge projects were undertaken, namely seL4 and the Verisoft project, which achieved outstanding results. Formal verification of OSes began to be adopted in the industry, mainly in the avionic domain. We give an overview of OS verification in Section 2.2. A comprehensive state of the art on OS verification until 2009 has been presented in [Kle09].

Hypervisor verification projects started to emerge shortly afterwards. The extensive use of hypervisors began in the early 2000. For example, Xen 1.0 was released after 2003, and VMware launched VMware workstation in 1999. Cloud services have known a spectacular success. Amazon Web Service, the leader for cloud services, whose solution is based on Xen, declared having more than 1 million costumers in 2015. Although similar, hypervisor and OS verification present some key differences, particularly when it comes to memory management. We present the state of the art of hypervisors in a separate section (Section 2.3).

Finally, in Section 2.4 we give details about the proof by abstraction methodology, that have been used in several projects, including ours. We show how our approach converge or diverge from the state of the art.

2.1 Early System Verification Projects

UCLA Secure Unix The UCLA Data Secure Unix OS is a general purpose operating system with verifiable security properties [Pop+79]. The OS was developed trying to keep the system's structure simple, they implemented it in Pascal, a high-level functional language. It supports the Unix interface, thus it provides a non-trivial set of system calls.

Their design is modular, and they separate the mechanism from the policy. More specifically, the kernel implements four abstract types: the processes, the pages, the devices and the capabilities. A set of operations is available for each type. For example, a capability might be granted or revoked. The policy then specifies which operation is allowed for a particular object. The policy is managed by a special process called the policy manager. In particular, only this process may grant a capability.

Walker et al used the proof by abstraction technique. They reported that the performance of their OS was finally poor, and that the proofs where tedious due to the unappropriated tools [WKP80] at this time. Even if they only reached 20 percent of their proof goals, their modeling and specification work allowed them to discover many flaws.

PSOS The Provably Secure Operating System is a formally specified tagged-capability hierarchical system architecture [NF03; FN79]. Hierarchical system architecture, or **layered architecture**, means that the system is built by layers, and that every layer is an implementation of the upper layer. In the uppest layer we find user abstraction, whereas the lowest layer contains the capabilities. In a layered architecture, a functionality is implemented in a level where irrelevant details for its implementation are not visible, i.e. in the highest level that allows to express it. Contrarily to the abstraction method, a level is not an abstraction of another level, a functionality is usually present in only one level. The goals of such an approach is to build a modular system, and to get rid of details that are not needed for reasoning at some levels. In PSOS, capabilities have a unique identifier and cannot be forged. This is enforced by the hardware: each capability has a tag unalterable by programs, so that hardware can recognize capabilities and forbid their modifications.

Feiertag and Neumann followed the Hierarchical Development Methodology, which involves a clear separation of the seven stages of realization. Each layer of the kernel is composed of one or a small number of modules. The five first stages of the methodology concern the conceptualization, the definition of the interface of each module, the formal specification of the modules and the formal representation of the data structure of each layer. The sixth stage concern the abstract implementation of a layer with the data structures of a lower level. The implementation into an executable program is done in the last stage. They wrote their specifications with the SPECification and Assertion language called SPECIAL.

Kit Bevier et al. developed the Kernel for Isolated Tasks to study the verification of the isolation of processes [Bev89]. Kit is small and very simple; there is no dynamic creation of processes or allocation of objects. Processes do not share memory, and protection of memory is not done through common virtualization technique but rather by attributing to each process a predefined segment in memory. Yet the task isolation property was entirely formally verified, using the Boyer-Moor theorem prover. To achieve this, they proved strong properties such as: the termination of kernel routines, the correctness of the address space abstraction, the isolation of the operating system from tasks, and that

the execution of the task is never done in supervisor mode. They were the first to prove the correct implementation of a complete OS.

2.2 Recent OS Verification Projects

At the border between early and recent verification projects, we can mention the Fiasco and the EROS kernels. The VFiasco project aimed at verifying parts of the Fiasco kernel, which stems from the L4 micro-kernel family [HT05]. They formalized the semantics for a part of C++, including some behaviors that are not specified by the standards, yet needed for their OS code.

The Extremely Reliable Operating System (EROS) is a capability based kernel designed for security [SH02]. The model has been verified (although manually), but the refinement to implementation level has not. Its successor, the Coyotos project was supposed to be verified at implementation level, and to reach the EAL7 certification. However the last publication date back from 2008.

The Integrity-178B kernel is a commercial OS for avionic purpose [Ric10]. It was developed by GreenHills and has reached an EAL6 advanced CC certification. They certificate was delivered on a previous version of the CC, which makes it difficult to compare with the CC as presented in Section 1.4. Basically, the specification model and the functional specification level of the Integrity-178B kernel are *formal* whereas the lowest model is *semi-formal*.

Similarly, the separation kernel PikeOS was developed for avionic application purposes, as part of the Verisoft XT project (see Section 2.3.2) [BB09]. They proved, on an abstract model, that there were no interference between the partitions [Bau+11]. Both Integrity-178B and PikeOS are ARINC 653 compliant.

Prove & Core is a microkernel like OS developed by Prove & Run [Les15]. The proof of isolation is almost completed at the time of writting. Although we proved a similar property on the memory, the memory management in an OS differs a lot from the memory management in a hypervisor.

In the section below, we describe the seL4 project. This kernel uses a capability system similar to EROS, and is from the L4 family, as is VFiasco.

2.2.1 SeL4

SeL4 is the first microkernel whose implementation functional correctness and security properties have been proved [Kle+09]. It constitutes a major step in the field of OS verification. The whole project took 29 person-years, among which 11 concern the proof of functional correctness of SeL4, and 4.1 the proof of the security properties.

The proof was conducted by abstraction. They proved the functional correctness from the binary executable to the most abstract state. The abstract state involves high-level structures. More models are developed on it, each one being designed for proving a particular property. We detail them below.

The protection state is the part of the global state which is responsible for managing the access rights. The protection state defines the authorities entities have over each other, and how entities might modify the protection state. The protection state is not static, and some authorities might be redundant, which render reasoning difficult. To get over these issues, they introduced an access control policy model which abstract the protection state into a static policy. Integrity was proved on this model [Sew+11].

Murray et al abstracted this policy into an information-flow policy and proved confidentiality on this model, which is a variant of intransitive non-interference [Mur+12; Mur+13]. The scheduler is allowed to flow to any partition, but no partition is allowed to flow to the scheduler. Otherwise the information flow property would be meaningless. Proving confidentiality implied determinising the system [DBK14a] and modifying the scheduler, it required 5 times the effort of proving integrity and authority confinement.

The properties of integrity and confidentiality differ from our properties. Their system is more generic than ours, the seL4 kernel is indeed highly configurable, thus the properties their proofs depend on the particular policy defined in the initial setup.

Our work is related to theirs because we used a similar approach, by successive refinements. However the order of magnitude of our work is not comparable to theirs. Our kernel is much smaller than seL4, we did not prove functional correctness down to the binary, and we only prove properties about the memory management part. In addition, the design of the hypervisor on which we work differs significantly from seL4. For example we do not have any notion of capabilities, whereas capabilities are at the core of seL4 access right enforcement. Finally, the SPT management is a feature of a hypervisor. Although seL4 can be used as a hypervisor, by combining it with seL4-VMM, the memory management in the VMM has not be proved to our knowledge.

2.3 Hypervisor Verification

Hypervisor verification projects are more recent, the oldest related work studied here is from 2009. The existing projects vary a lot in the methodology used, the properties targeted and the maturity of the project. Yet interestingly, many of them have published about the memory management, which is considered as a critical part of the system.

The eXtensible and Modular Hypervisor Framework (**XMHF**) is a small open source hypervisor supporting one guest. Vasudevan et al [Vas+13] proved that the guest cannot write in hypervisor memory, i.e. they proved the integrity of the hypervisor memory. They verified some modules of the hypervisor automatically, using the CBMC model checker, and others manually, due to the limitation of the tool. Andrabi extended the automatic verification by proving the well-formedness of the PT setup in [And13]. They do not virtualize the memory with SPTs, but rather use the hardware virtualization solution. The use of a model checker makes their model and methods different as when using a theorem prover. Furthermore, as they consider one guest, the properties targeted are inherently different from what we want to ensure for several guests. Indeed, the non-interference or confidentiality property do not make sense, neither does the integrity of the guest memory.

Blanchard et al. have presented a case study on the creation of a new mapping in a PT [Bla+15] on the **Anaxagoros** hypervisor. They used the Frama-C framework to conduct their proof. They annotated their C code with pre/post conditions in ACSL and the goals were discharged to automatic theorem provers. In contrast to us, they considered parallelism and showed that their model was valid for weak memory models. The method and goals are thus quite different. They worked on a part (i.e. one function) independently of the rest of the system whereas we modeled the interactions between the several parts of the system, to prove high-level properties on the whole system.

Barthe et al. formalized an idealized model of a para-virtualized hypervisor in Coq [Bar+12]. They included the caches in their model and considered cache-based side-channel attacks, which is out of our scope. On the other hand they make several simplifications, such as considering only one level of page tables or not considering any sharing

between guests. Our works are not directly comparable, because they do not refine their model to an implementation level. In particular, they use abstract data structures to represent PTs, thus the PTs are not included in their attack surface. Whereas a great part of our invariants concerns the well-formedness of the PTs structure.

We develop in more details the PROSPER and the Verisoft XT project, which are more related to our work. The latter is certainly the project in which the SPT management has been the most extensively studied.

2.3.1 Prosper

Prosper is a separation kernel for ARMv7 developed by Dam et al [Dam+13]. It can run two guests and provides isolation of their component resources while enabling them to communicate through a communication line. Our isolation property is similar to theirs.

Their proof is carried mostly with the HOL4 prover, they work on top of the ARMv7 model developed in Cambridge, extended with a MMU model, on which they proved isolation properties when PTs are the identity map [KSD13]. They proved properties at machine-code level, with the binary analysis platform BAP [DGN13]. Some of our axiom on the ARM behavior are proved lemmas in their model.

They conduct their proof by abstraction. In their *ideal model*, the two guests execute on abstract separated ARMv7 platforms which communicate by asynchronous message passing. The proved the bisimulation between the ideal and concrete models.

Nemati et al. extended the HOL4 ARMv7 Cambridge model with an MMU, and proved integrity and confidentiality [NGD15]. Their kernel uses the direct paging mechanism, more precisely, the guest can manage its PTs through the hypervisor with 9 hypercalls. Thus the invariants to be proved to ensure isolation are quite different.

Our abstract model is much more abstracted than their ideal system presented in [Dam+13; Nem+15]. Regarding the memory management, they have abstracted the PTs of the hypervisor in their ideal model [Nem+15]. Whereas we have no notion of PTs anymore in our abstract model. Still, our methods and targeted properties are similar. In addition, we use the same ARMv7 platform, with one processor, we do not model caches, and have similar assumptions, e.g. the guests partitions are static. This makes our concrete transition systems akin.

2.3.2 Verisoft XT

The Verisoft XT project started in 2007 and ended in 2010, it is the successor of the Verisoft project. The goal was the creation of methods and tools which would enable the *pervasive* formal verification of computer systems. Their verification is pervasive as the properties are meant to be proved from the application level to the hardware. The project was split into two main parts, the OS project and the hypervisor project.

The Hypervisor project aimed at proving property on the Microsoft Hyper-V hypervisor [LS09]. Yet the majority of their published achievements concern two other hypervisors: a smaller version of the Hyper-V (baby hypervisor) on which a virtualization correctness proof was reported [Alk+10], and a academic prototypical hypervisor [Kov13].

On the academic hypervisor, Alkassar et al. have proved properties of correctness of the TLB virtualization mechanism [Alk+12]. All the details can be found in the PhD thesis of Kovalev [Kov13]. They proved that if a translation is present in the virtual TLB (i.e. the TLB that the guest would have if it were running directly on the hardware), it is also present modulo some translation stages in the hardware TLB (i.e. the cache of the SPT). They precisely modeled the hardware and formally abstracted it with reduction

proofs. They considered a concrete hybrid state, composed of a hardware and a software state. They showed a correspondence between the memory part of the virtual hardware state (i.e. the virtual TLB) and the memory part of the hybrid state (i.e. the TLB and the SPT algorithm). They stated their property for any SPT algorithm that would respect some properties. Then they designed a SPT algorithm and showed that this particular algorithm respects the established properties.

Their hardware model is more complex and realistic than ours: they consider caches, TLBs and store buffers. In addition, they consider a multi-processor hardware. However the particular SPT algorithm that they choose to instantiate the property is less realistic than ours. Our method and our focus points differ from theirs. Basically because our goals are different, the property we target is orthogonal to theirs. For example when it comes to isolation, cache correctness is less important. Indeed it is acceptable that the TLB should not be correctly managed as long as the TLB is flushed between each guest execution or distinct for each guest. Also, they suppose that there is always a free SPT slot when allocating a new one, because it is not critical for correctness. Whereas we go in deeper details in the modeling of the SPT allocator, as we consider that the proof of its well-formedness is a key aspect of the isolation proof.

2.4 The Methodology of Proof by Abstraction

The seL4, Prosper and Verisoft XT projects use proof by abstraction technique to prove properties on the system. We use this methodology as well. We have presented the main lines of OS and hypervisor verification projects in the previous section, this section aims at presenting the methodology, and at comparing our approach to the state of the art.

The goal of the proof by abstraction is to establish a correspondence between an abstract system specification and a concrete system implementation, such that properties of the abstract level can be reported to the concrete level. Indeed, many operations and complex algorithms are present in the concrete level only for performance and hardware compliance purposes. Yet these implementation details are not useful when it comes to reasoning. PTs are a good example, we use them to speed up the access to memory, and because it allows to make use of the security mechanism provided by the hardware MMU. Yet we phrase our specifications in terms of reachable regions of memory, the concept of PTs can therefore be abstracted.

Usually, one use the notion of *observable state* to compare the behaviors of a concrete and an abstract system. As its name suggests, the observational state is a partial view of a state, which ignores the notions which are not needed for the targeted properties. Intuitively, in our case, the observational state corresponds to what a guest can observe. The abstract and concrete states are projected on an observable state, and, roughly speaking, we verify that if a concrete and an abstract state verify a certain relation, they have the same observable behavior.

In our case, the observable states are the abstract states, and we do not use a relation between abstract and concrete states, but a function from concrete to abstract states. Our approach is akin to the second approach taken by Daum et al. for strengthening the proofs of seL4 [DBK14b].

The preservation of trace-based properties by refinement has been extensively studied in the literature [LV95]. However non-interference like properties, such as confidentiality, are not trace properties, and are not preserved by refinement. In the following, we explain briefly the formal link we establish between our two models, the kind of properties that we prove and why they are preserved by refinement.

2.4.1 Commutation

We consider two state transition systems, introduced in the following Definitions 2.4.2 and 2.4.3.

Definition 2.4.1 (Concrete State Transition System)**.** Let St be the concrete state space, $\rightarrow \in St \times St$ the concrete transition relation and $\mathcal{I}_c$ and $\mathcal{F}_c$ the sets initial and final concrete states, the concrete transition system is the tuple $(St, \mathcal{I}_c, \mathcal{F}_c, \rightarrow)$.

Definition 2.4.2 (Well Formed Concrete States)**.** $\widehat{St}$ is the subset of St containing only the well-formed states:
$\widehat{St} = \{st \in St | wf(st)\}$

Definition 2.4.3 (Abstract State Transition System)**.** Let $St^\#$ be the abstract state space, $\dot{\rightarrow} \in St^\# \times St^\#$ the abstract transition relation and $\mathcal{I}_A$ and $\mathcal{F}_A$ the sets initial and final abstract states, the abstract transition system is the tuple $(St, \mathcal{I}_A, \mathcal{F}_A, \dot{\rightarrow})$.

The two transition systems are in correspondence if they commute (Definition 2.4.5) regarding the $view$ function (Definition 2.4.4).

Definition 2.4.4. $view : St \rightarrow St^\#$ is the abstraction function, which is total on $\widehat{St}$.

Definition 2.4.5 (Commutation)**.** $\forall st_1, st_2 \in St$, if a transition is possible from st_1 to st_2 [1], if the $view$ function is defined on st_1 [2], then a transition from the view of st_1 is possible [3] and equal to the view of st_2 [4].

If [1] $st_1 \rightarrow st_2$
[2] $view(st_1) = {st^\#}_1$
Then [3] ${st^\#}_1 \dot{\rightarrow} {st^\#}'_2$
[4] $view(st_2) = {st^\#}'_2$

Note that we also prove that each transition from a well-formed state preserves its well-formedness. The $view$ function being total on $\widehat{St}$, the points [2] and [4] always succeed when $st_1 \in \widehat{St}$.

2.4.2 Transferring Properties to the Concrete Model

We present below our two properties, integrity and confidentiality, and show how they can be transferred to the concrete level.

Integrity Our property of integrity states that, if a guest is not currently running, then a transition does not change its secret data. The current guest is the same in the concrete and the abstract state, in other words, the $view$ function restricted to the field $curr$ is the identity.

The integrity property that we prove on the abstract level is defined in Definition 2.4.6. If the current guest is not i, then some field stay unchanged. We write x_i for one or several fields that should not be modified by the transition of i, we will specify to which fields such a x_i corresponds in Chapter 5, at the moment we are just concerned with the transfer of such properties.

Definition 2.4.6 (Integrity at Abstract Level)**.**

$$int^\#(St^\#, \dot{\rightarrow}) \Leftrightarrow \forall a_1, a_2 \in St^\#, (a_1.curr \neq i \wedge\ a_1 \dot{\rightarrow} a_2) \Rightarrow (a_1.x_i = a_2.x_i)$$

The integrity property at concrete level is defined in Definition 2.4.7. As can be seen, we do not express it in terms of unchanged fields of the concrete state. Indeed, we want to express what has not been modified in terms of observable state. Of course, if one is not convinced that such an abstract property has a meaning on the concrete model, one could prove that an equality of a field in the abstract model implies the targeted equalities on the concrete model. However that would mean expressing the property on the concrete level, which is precisely the thing we try to avoid by using the abstraction method.

Definition 2.4.7 (Integrity at Concrete Level)**.**

$$int(St, \rightarrow) \Leftrightarrow \forall c_1, c_2 \in St, (c_1.curr \neq i \wedge\ c_1 \rightarrow c_2) \Rightarrow (view(c_1).x_i = view(c_2).x_i)$$

What we want to prove now is Lemma 2.4.1, which states that if the property of integrity is proved on the abstract state then it implies the integrity property on the concrete state. The proof follows immediately from the definitions.

Lemma 2.4.1 (Transfer of Integrity)**.** $int^{\#}(St^{\#}, \dot{\rightarrow}) \Rightarrow int(\widehat{St}, \rightarrow)$

Proof. Let $c_1, c_2 \in \widehat{St}$, such that $c_1.curr \neq i$ (1) and $c_1 \rightarrow c_2$ (2).

From the definition of commutation (Definition 2.4.5), $view(c_1) \dot{\rightarrow} view(c_2)$.

In addition, from (1) we have that $view(c_1).x \neq i$, because $view$ is the identity on the field $curr$.

Therefore, from $int^{\#}$, $view(c_1).x = view(c_2).x$ (Definition 2.4.6), Qed. □

Confidentiality Confidentiality means that the execution of a guest does not depend of *secrets* of other guests. We formalize this by stating that if two guests are equals modulo some so-called secret fields, and if the guest running is not supposed to have access to these secrets, then the executions from the two similar states end up in two states equal on some fields. It means that the execution has the same impact on these fields, whether the secrets are equal or not.

In the property of confidentiality of abstract states described below, we state that if two states are equal on some fields, represented by y, then, after the execution, they are equal on some fields, represented by z. Note that if we want the property to be transitive, we need y to be equal to z. Again, y and z may depend on the particular guest i, hence the notation y_i and z_i.

In reality, our property is slightly more complicated because some regions of memory are shared, so the execution of a guest might depend on them, we develop in Chapter 5.

Definition 2.4.8 (Confidentiality for Abstract State)**.**

$$\begin{aligned} confid^{\#}(St^{\#}, \rightarrow) \Leftrightarrow\ & \forall a_1, a_2, b_1, b_2 \in St^{\#}, \\ & (a_1.curr \neq i, \\ & b_1.curr \neq i, \\ & a_1.y_i = b_1.y_i, \\ & a_1 \dot{\rightarrow} a_2, \\ & b_1 \dot{\rightarrow} b_2) \Rightarrow \\ & a_2.z_i = b_2.z_i \end{aligned}$$

A fundamental property of our abstract system is *determinism*. In our transition system, confidentiality can only be proved because of determinism. Indeed, if several transitions are possible from a state a_1, e.g. $a_1 \dot{\rightarrow} a_2$ and $a_1 \dot{\rightarrow} a_3$, we do not even have, in the

general case, the property that $a_2.z = a_3.z$. Non-determinism is usually the reason why non-interference properties are not proved by refinement. But with our deterministic system, the proof of transfer is straightforward.

Definition 2.4.9 defines the confidentiality at concrete level. Again, we express the notion of similarity in the abstract space.

Definition 2.4.9 (Confidentiality for Concrete State)**.**

$$\begin{aligned} \mathit{confid}^{\#}(St, \rightarrow) \Leftrightarrow \; & \forall c_1, c_2, d_1, d_2 \in St, \\ & (c_1.\mathit{curr} \neq i, \\ & d_1.\mathit{curr} \neq i, \\ & \mathit{view}(c_1).y_i = \mathit{view}(d_1).y_i, \\ & c_1 \dot{\rightarrow} c_2, \\ & d_1 \dot{\rightarrow} d_2) \Rightarrow \\ & \mathit{view}(c_2).z_i = \mathit{view}(d_2).z_i \end{aligned}$$

Finally, we prove the Lemma 2.4.2.

Lemma 2.4.2 (Transfer of Confidentiality)**.** $\mathit{confid}^{\#}(St^{\#}, \dot{\rightarrow}) \Rightarrow \mathit{confid}(\widehat{St}, \rightarrow)$

Proof. Let $c_1, c_2, d_1, d_2 \in \widehat{St}$, such that $c_1.\mathit{curr} \neq i \wedge d_1.\mathit{curr} \neq i$ (1), $c_1 \rightarrow c_2$ (2), and $d_1 \rightarrow d_2$ (3).

From the definition of commutation (Definition 2.4.5), $\mathit{view}(c_1) \dot{\rightarrow} \mathit{view}(c_2)$, and $\mathit{view}(d_1) \dot{\rightarrow} \mathit{view}(d_2)$.

In addition, from (1) we have that $\mathit{view}(c_1).x \neq i$, and $\mathit{view}(d_1).x \neq i$ because view is the identity on the field curr.

Therefore, from $\mathit{confid}^{\#}$, $\mathit{view}(c_2).z = \mathit{view}(d_2).z$ (Definition 2.4.8), Qed. □

2.4.3 Comparison of our Abstraction to State of the Art

Proof by refinement is popular in OS verification. The novelty of our approach though, is that we abstract entirely the structure of PT, while still enabling to reason precisely on memory.

As we will see in more details in Chapter 4, proving that two PTs map two areas of memory isolated one from each other rely on the proof that each of these PT is well-formed and does not map itself nor the other PT with unprivileged rights. Once we have proved these properties on PT, we can interpret PT as functions from virtual to physical addresses, so we do not need to keep the detail of their structure. That is why we abstract their structure, i.e. we represent memory as mappings from addresses to a pair of byte and rights. Another solution could be to keep mapping functions from virtual to physical addresses with rights, if needed, but the important point is that the PT structure is not needed and thus should be abstracted.

As we have briefly mentioned in this state of the art, the abstraction presented by Dam et al. is a duplication of a ARMv7 processor. They have proved that a hypervisor running two guests on one ARMv7 processor is equivalent as having two guests running on two separate processors [Dam+13]. Their abstract system stays close to the implementation level. In particular, they keep the complexity of PTs of the guest, but the PTs of the hypervisor are abstracted away [Nem+15]. Therefore, they have abstracted away the PTs whose structure is managed by the hypervisor.

Daum et al also take this approach of keeping the PT in their abstract model (which is not their most abstract level). In their model, threads have capabilities to virtual memory

objects, such as PTs and pages. They abstract the objects in their access control policy and information-flow policy. More precisely, they group several objects under some labels and express their policies on these groups. Therefore, their highest levels are very abstract, because there is not any notion of PT, not even memory. Their proof of well-formedness of their PTs is part of the proof of well-formedness of their capability space, thus abstracting the PT in their abstract domain would not make much sense, therefore, even if our approaches shares similarities, they cannot really be compared on this point.

Our approach allows to have a model where memory is detailed enough to be reasoned about, for example, we could still reason at fine granularity about copy in this kind of model, while abstracted enough to ease the reasoning.

2.5 Contributions

The notion of "proving properties on a system" can be broadly interpreted. In particular, in the OS domain, the systems considered are large and complex, and the proof are done on models. Therefore it is not always obvious to figure out the assurance provided by a proof. We identify below three characteristics which allow to assess the trust we can have on a system.

Confidence in the meaning of the properties. The first problem to raise is the difficulty to express meaningful properties on large and complex systems. It is all the more true that the properties are high-level (e.g. confidentiality), and that the system is low-level. For example, as we have mentioned earlier, our properties would be hard to express and to understand if they were stated on the concrete level. We resort to abstract models to increase the confidence we have that the properties stated have the intended meaning. We can see, moreover, in the the state of the art just presented, that most of the projects prove their properties on an abstract model. Some of the abstraction stay very close to the concrete model (e.g. in the Prosper project), other are more abstract (e.g. the idealized model of Barthe et al.). The more abstract the model is, the greater assurance one get that the property has the intended meaning.

Confidence in the correspondence between the model and the implementation. If the model does not correspond to the implementation, then the properties proved are null and void. Therefore, contrarily to the previous point, we could say that the closer the model is from the implementation, the more we can trust our model to comply to the implementation. As we have explained in Section 2.4, in order to take the best of both world, we can prove a refinement between an abstract and a concrete model. This solution enables to maximize the trust in the proofs. This is the solution that we have taken. In particular, the tools developed by Prove & Run allow to generate C code from the concrete model, thus allowing to strengthen the confidence that the model on which the proofs are made corresponds to the executed code. On the other hand, some projects mentioned in the state of the art consider only one model, or consider several models but do not prove the refinement between them. Therefore, they face a trade off between gaining trust in the meaning of the properties and gaining trust in the fact that the properties actually hold on the implementation level. In particular, these models cannot be abstracted too much.

Confidence between the implementation and the execution. If a compiler is bugged, or if the implementation does not comply to the standard of the language, the compiled program may have a different semantic from the initial program. Some tools allow to establish a formal link between the binary and the code. For example the verified C compiler CompCert verifies that, for a subset of C, under some assumptions, the semantic of the code is preserved [Ler09]. This compiler can be used to compile some parts of an OS, but can hardly be used for the compilation of a whole OS, because an OS usually use code outside of the subset of C on which the theorems of CompCert are valid [Kle09]. Another example is the Binary Analysis Platform, which was used by Dam et al on certain parts of the hypervisor code ([DGN13; Bru+11]). This aspect was out of the scope of our study.

Finally, note that the semantic of the binary depends itself on the machine. However, the behavior of the machine may not comply to its specification. None of the systems cited in the state of the art run on proved hardware machines, indeed as the platforms are widely used and therefore tested, they are usually trusted. We use ARMv7, and we only use classical features of the board, in particular we do not use the virtualization extensions. Therefore, we consider that we have reasonable assurance that the behavior of the hardware corresponds to its specification.

Our first main contribution is the proof, at a high level of trust, that the isolation of the memory of the guests is ensured by a SPT algorithm which is realistic and does not present particular simplifications. We claim a high level of trust in the properties we prove, because our methodology apply the two first confidence principle just mentioned. In particular, our abstract model has no more notion of PTs, while keeping the information they provide. The step between our two models is huge, we reach a very high-level model while formally linking it to a low-level close to the C code.

Our second main contribution is methodological. In the literature, various books for best practice in software development or software testing can be found. They are the result of many years of experience feedback, from the academic and from the industry. Formal proof is not as widespread as software development or testing, especially in industry, where the proof of large projects is almost non-existent. In this thesis, we present with many details our methodology. We present the properties we need and how they are connected. We expose the difficulties we encountered and the limit of our proofs.

Our work has lead to a publication in the FASE conference [BJS16].

2.6 Overview of the Chapters

We give an overview of the steps needed to achieve the proof, and in which chapter they are developed.

The hypervisor manages the memory and the privileged registers, and manages guests access to them. It provides virtualized resources to the guests, so that they can execute as if they were running on bare metal. As depicted in Figure 2.1, the state of the system can be decomposed in two parts:

- The hardware state, which captures the state of the physical memory and registers.
- The hypervisor state, which captures the virtualized machine state for each guest.

The state of the hypervisor represented in the upper part of the Figure is only constituted by software. Ellipses in Figure 2.1 show the resources that may be modified by the execution of each guest. Roughly speaking, a transition of the system can be decomposed as such:

1. The guest executes, changing only the part of the *hardware state* that it can access.
2. The hypervisor handles a hypercall or exception for this guest, modifying the *virtualized registers* for this guest, and the *hardware state*.

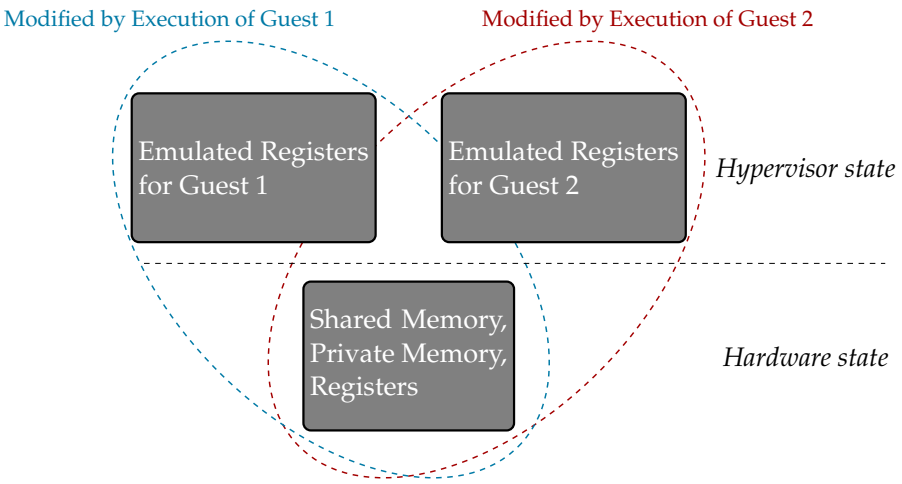


FIGURE 2.1: Modified Resources during Concrete Guest Execution

[↪ *The concrete transition system is described in Chaper 3*]

Our goal is to prove that, even if guest executions modify common resources, none of the guests can tamper with the execution of other guests. For example, no guest should be able to access the private memory of another guest.

Isolation of memory is ensured by the the mechanism of PTs. The physical addresses reachable by one guest are the one mapped by its Shadow Page Tables (SPTs). The primary condition to reason about PTs is to prove that they are well-formed, and that no transition break these properties. The hypervisor also makes sure that other properties holds, such that the memory locations reachable through the SPTs of one guest are allowed locations for this guest. We verify that these properties are preserved by transitions of the system.

[↪ *Proof of invariant properties over the concrete system are presented in Chapter 4*]

Showing that the memory locations reachable through the SPTs are allowed is only the *first part* of the proof. The *second part* is to prove that this property implies isolation. To do so, we show the correspondence between the concrete level and an abstract, idealized system on which isolation is straightforward.

The correspondence is formally proven, following the method presented in Section 2.4. We define an abstraction function and abstract transitions, and we prove the commutation of each concrete transition with its counterpart abstract transition. The refinement proof relies on properties of the concrete system established by the invariants.

[↪ *The refinement proofs are presented in Chapter 5*]

It is difficult to reason on the concrete level, due to the complexity of the PT structure, and to the fact that every modification of the memory might modify the SPT in a non wanted way, as SPTs are located in memory. On the other hand, the proof of isolation is straightforward on the abstract system. The Figure 2.2 depicts such an isolated abstract system. Contrarily to the precedent Figure 2.1, only explicitly shared resources may be altered during guest execution.

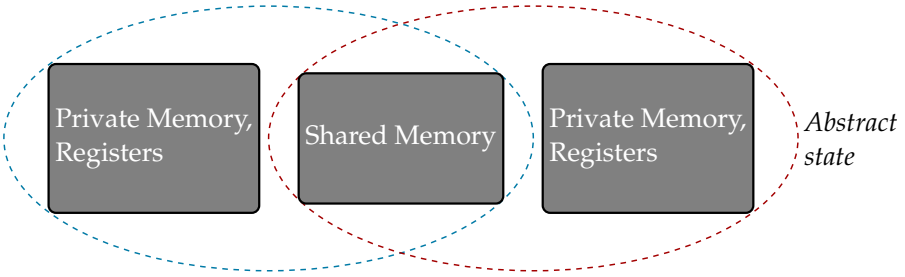


FIGURE 2.2: Modified Resources during Abstract Guest Execution

[↪ *The abstract transition system and the proofs of isolation are presented in Chapter 5*]

2.7 Key Points

- OS and hypervisor verification grew in importance 15 years ago.
- Several projects have studied the memory subsystem, which is complex and error-prone.
- The SPT has been extensively studied in the Verisoft XT project, but with different goals and methodology.
- The PROSPER hypervisor verification project targets similar properties to ours, however they do not use SPTs for virtualization.
- The complete abstraction of SPTs as we have done has not been seen elsewhere. It simplifies drastically the model, and enables to focus on high-level properties.

Chapter 3

Concrete Model of the Hypervisor

Contents

In this chapter, we present the concrete transition system of the hypervisor. We first detail how we represent the Page Tables (PTs). Then we go through our modeling, the assumptions we make and the characteristics of the hypervisor. Finally, we present the transitions of our system.

This chapter is particularly dense and introduces many notations. For the reader in a hurry, we propose a shorter path of reading:

- Definition of PTs: Definition 3.2.10.
- Static permissions: Section 3.3.1.
- Definitions of the concrete state: Definitions 3.4.2, 3.4.6 and 3.4.7, without paying attention to register fields.
- Guest transition: Section 3.5.1.
- Memory Management transitions: Section 3.5.3.

3.1 Basic Types and Notations

Before presenting our concrete model, we introduce in this section the basic types and notations that we use afterwards:

- $Uint$ represents unsigned 32 bits integers.
- Reg represents 32 bits registers.
- $Addr$ represents 32 bits addresses. Note that we model both *virtual* and *physical* addresses by $Addr$.
- $Addr_{pg}$ denotes the addresses aligned to the size of a page (i.e. addresses that are multiple of the size of a page). $Addr_{pg}$ is a subtype of $Addr$. As we will detail in Section 3.2, we work with pages of size 2^{12}. $Addr_{pg}$ thus represents the addresses whose 12 least significant bits are equal to zero.
- Let $a, b \in Uint, Reg$ or $Addr$, $a + b$ is an addition modulo 2^{32}. Furthermore, the addition of an $Addr$ and an $Uint$ is defined: $+ : Addr \times Uint \rightarrow Addr$ and also corresponds to an addition modulo 2^{32}.
- $Byte$ represents bytes, and $Byte^n$ represents a sequence of n bytes.
- Mem represents the memory, it is a function from addresses to bytes: $Mem = Addr \rightarrow Byte + Oob$. The function returns Oob for addresses out of its boundaries.
- If X is a type, $\{X\}$ denotes a set of elements of type X, $list < X >$ denotes a list of elements of type X.
- $\forall l \in list < X >, \forall A \in \{X\}$, the notation $l \subset A$ means that all the elements of the list l are in the set A.
- When a list is not empty, we write $< head :: tail >$ for its decomposition into a head and a tail.
- $\forall x, y \in Uint$ (resp.$Addr$), $[x, y[= \{x \in Uint$ (resp.$Addr) \mid a \leq x < b\}$. Just as for the lists, we denote by $\subset$ the inclusion of an interval in a set.
- We use the symbol "_" as a wildcard. For example "if $f(a, \_) = c$" means "if $\exists b, f(a, b) = c$".
- The notation $\begin{cases} a_1 & : Type_1 \\ ... & \\ a_n & : Type_n \end{cases}$ is equivalent to $(a_1 : Type_1 \times ... \times a_n : Type_n)$. We use it for clarity's sake.
- In the remaining of the document, we will define many records, the notation $a.x$ means that we access the field x of the record a.

3.2 Modeling of the Page Tables

PTs are at the core of our study and proofs. We present in this section our modeling of their structure.

A PT maps *virtual* addresses to *physical* addresses and provides the access rights to each physical address it maps. We denote the set of rights by $Rights$, it contains three elements:

$$Rights = \{pl1, rw, ro\}$$

$pl1$ means Privileged Level 1, it means that the address is accessible only in *privileged* mode, i.e. accessible only by the hypervisor. rw means that the address is readable and writable in *unprivileged* mode, ro means that the address is only readable in *unprivileged* mode. We define a total order relation "$\geq$" over $Rights$ by $rw \geq ro \geq pl1$.

A PT function has the following type:

$$PT = Addr \to (Addr \times Rights) + Fault$$

A virtual address $va \in Addr$ whose image is in $Fault$ corresponds to an address for which there is no translation, i.e. the access to va would raise a fault.

Definition 3.2.1. (Mapped to/Mapped by) Let $table \in PT$,

1. Just as for any function, we say that a virtual address va is *mapped to* a physical address pa when $table(va) = (pa, \_)$, and that va is not mapped if $table(va) \in Fault$.
2. We say that a physical address pa is *mapped by* a virtual address va in the PT $table$ with some rights r when $table(va) = (pa, r)$.
3. More generally, we say that a physical address pa is *mapped by* the PT $table$ when there exists $va \in Addr$ such that $table(va) = (pa, \_)$. Otherwise it is *not mapped*.

The function pt takes a memory and a pointer to a PT and returns the PT located there. $pt(mem, base_{\mathrm{PT}})$ is read as "the PT at address $base_{\mathrm{PT}}$ in memory mem":

$$pt : Mem \to Addr \to PT$$

The function pt is a composition of several functions, we introduce them below, before giving the definition of pt.

3.2.1 Decomposition of the Function pt

We work with two levels of PTs, as illustrated in Figure 3.1. First and second level PTs are arrays of descriptors. More specifically, we use ARMv7 short descriptors, and we only use small pages of 4KB . A first level PT contains 4096 descriptors of 4 bytes each, whereas the second level PT contains 256 descriptors of 4 bytes each. A descriptor of a first level PT contains the base address of a second level PT [1]. Whereas a descriptor of a second level PT contains the address of a page and rights related to it. In both cases, the descriptor might not lead to anything, in this case it is a fault. The definition of descriptors is given in Definition 3.2.2.

Definition 3.2.2 (Descriptors Types)**.**

$$Descr_1 = Addr + Fault$$

$$Descr_2 = (Addr \times Rights) + Fault$$

[1] A descriptor of a first level PT can also contain the base address of a *section*, which is a page of size 1MB. However, we do not use sections, therefore we do not model them

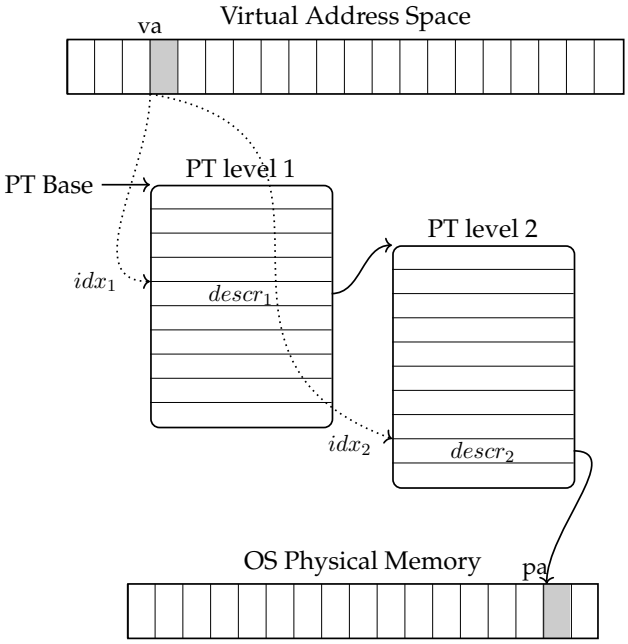


FIGURE 3.1: Page Table Walk

To read PTs in memory, we use functions that interpret a sequence of 4 bytes into a descriptor.

Definition 3.2.3 (Bytes to Descriptors)**.** For $i \in \{1, 2\}$:

$$descr_i : Byte^4 \rightarrow Descr_i$$

The bytes are fetched from memory by the following function, which returns Oob if the range of bytes requested is out of memory boundaries.

Definition 3.2.4 (Fetch)**.**

$$fetch : Mem \times Addr \times (size : Uint) \rightarrow (Byte^{size} + Oob)$$

Definition 3.2.5 (Domain of Fetch)**.** For all addresses $a \in Addr$, $mem \in Mem$ and size $s \in Int$, if $a + s < size_{mem}$, then $fetch(mem, a, s)$ returns a sequence of bytes:

$$a + s < size_{mem} \Rightarrow fetch(mem, a, s) \notin Oob$$

For readability, we define a third function for each descriptor, $get\_descr_i : Mem \times Addr \rightarrow Descr_i + Oob$ which combine $fetch$ and $descr_i$:

Definition 3.2.6. (Get Descriptor) $\forall mem \in Mem, \forall a \in Addr$,
$\boldsymbol{get\_descr}_i(mem, a) =$
If $fetch(mem, a, 4) \in Oob$
Then **return** oob
Else **return** $descr_i(fetch(mem, a, 4))$

To translate a virtual address va into a physical address, va is decomposed into three parts: an index in the first level PT, an index in the second level PT, and an offset in the page. More precisely:

- The first index corresponds to the 12 most significant bits of the 32 bits address va. This index allows to access the 4096 entries of the first level PT ($2^{12} = 4096$).
- The second index corresponds to the 8 following bits of va. It allows to access the 256 entries of the second level PT ($2^8 = 256$).
- The offset corresponds to the 12 least significant bits of va. It allows to access the 4096 bytes of a small page.

Definition 3.2.7 (Sizes)**.** We write $size_{PT1}$ for 16KB, the size of a first level PT ($4 * 4096 = 16384$); $size_{PT2}$ for 1KB, the size of a second level PT ($4 * 256 = 1024$); and $size_{page}$ for 4KB, the size of a page.

Definition 3.2.8 (Virtual Address Decomposition)**.** Forall $va \in Addr$, we note

$$va = idx_1 \oplus idx_2 \oplus \mathit{off}$$

the unique decomposition of va into two indexes idx_1, idx_2, and an offset off.

The following property follows trivially:

Property 3.2.1 (Decomposition Properties)**.** For all $va \in Addr$, for all $idx_1, idx_2, \mathit{off} \in Uint$ such that $va = idx_1 \oplus idx_2 \oplus \mathit{off}$"

- $idx_1 < 4096 \wedge idx_2 < 256 \wedge \mathit{off} < 4096$
- $\mathit{off} = 0 \Rightarrow va \in Addr_{pg}$
- Forall $va' \in Addr, idx'_1, idx'_2, \mathit{off} \in Uint$ such that $va' = idx'_1 \oplus idx'_2 \oplus \mathit{off}'$:
  - $idx_1 = idx'_1 \wedge idx_2 = idx'_2 \Leftrightarrow va$ and va' are in the same page

Finally we denote by $compute$ the function that computes the address of a descriptor in a PT, given the index of the descriptor and the base address of the PT:

Definition 3.2.9 (Compute)**.** Multiplies the size of a descriptor by the index idx of the descriptor and adds it to the $base$ address.

$$compute : (base : Addr) \times (idx : Int) \rightarrow Addr$$

We can now give the definition of the PT function.

Definition 3.2.10 (Definition of the Page Table Function)**.** $\forall mem \in Mem, \forall base \in Addr, \forall \boldsymbol{pt} \in PT, \forall \boldsymbol{va} \in Addr$,

$\boldsymbol{pt}(mem, base)(\boldsymbol{va}) =$
Let $\boldsymbol{idx}_1, \boldsymbol{idx}_2, \boldsymbol{off} \in Uint$ such that $\boldsymbol{va} = idx_1 \oplus idx_2 \oplus \mathit{off}$
$base_2 = get\_descr_1(mem, compute(base, \boldsymbol{idx}_1))$
If $base_2 \notin \{Oob, Fault\}$
Then
  Let $descr_2 = get\_descr_2(mem, compute(base_2, \boldsymbol{idx}_2))$
  If $descr_2 \notin \{Oob, Fault\}$
  Then
    Let $(\boldsymbol{pa}, \mathbf{r}) = descr_2$
    return $(\boldsymbol{pa} + \boldsymbol{off}, \boldsymbol{r})$
  Else **return** ***fault***
Else **return** ***fault***

Less formally, $pt(mem, base)(va) = (pa, r)$ means that the physical address pa is mapped with rights r by the virtual address va in the PT of base address $base$ in the memory mem. Similarly, $pt(mem, base)(va) \in Fault$ means that the address va is not present in the PT at base address $base$ in memory mem.

Every modification on the memory might change the mappings defined by a PT. When the modification is performed on a region of the memory distinct from the regions where we keep the PTs (the *pools*), we easily prove that the PTs stay unchanged. However when we modify the region where the PTs are kept, in order to add or remove a mapping in a PT for instance, it is more difficult to show which parts of the PTs are affected and how by the modification. The proof rely on many properties of well-formedness of the PTs, that we will present in Chapter 4.

3.2.2 Virtual Page Table Walk

Once the MMU is activated, the memory can only be accessed with *virtual* addresses. Indeed, the MMU would translate any address to a physical address before accessing it, using the PTs. Consequently, when the hypervisor runs, it only addresses memory through virtual addresses.

The PTs are maintained by the hypervisor. It means that the hypervisor needs to access the entries of the first and second level PTs, in order to read or modify them. However, we have seen in Section 3.2.1 that the PT is composed of arrays whose entries hold addresses of other arrays or pages, these addresses are *physical*. For example, in Figure 3.1, the PT base pointer is a physical address, and $Descr_1$ and $Descr_2$ hold physical addresses too. Suppose that the hypervisor needs to modify $Descr_2$. It uses the base address of the first level PT and the index idx_1 to compute the *physical* address of $Descr_1$, which must be translated to a virtual address before being accessed. The $Descr_1$ holds a second level PT physical address, which allows to compute the physical address of a $Descr_2$. The hypervisor would need again to translate it to a virtual address before accessing it.

Therefore, the hypervisor uses a function $phys2virt$, different for each guest (Definition 3.4.7), in order to translates physical addresses to virtual addresses before accessing them. This function is the reciprocal of the PT on the $pool$ addresses where the latter is defined. This is ensured by the two invariants described in Section 4.1.2.

Furthermore, the hypervisor cannot call the function get_descr_1 and get_descr_2, which use physical addresses. Theses functions are defined for specification purpose only. We define the two functions which model the access to descriptors with virtual addresses made by the hypervisor.

The versions of get_descr_1 and get_descr_2 with virtual addresses take one more parameter: a PT address. They model the behavior of the hardware when the hypervisor access a descriptor. Basically, the virtual getters translate the virtual address to a physical one with the PT and then call get_descr_1 or get_descr_2. We give the formal definition of a virtual getter below.

Definition 3.2.11 (Virtual Get Descriptor). $\forall mem \in Mem, \forall base_{\mathrm{PT}} \in Addr, \forall \boldsymbol{va} \in Addr$:
$\boldsymbol{get\_v\_descr_i(mem, base_{\mathrm{PT}}, va)} =$
If $pt(mem, base_{\mathrm{PT}})(\boldsymbol{va}) \notin Fault$
  Let $pa \in Addr$ s.t.$(pa, \_) = pt(mem, base_{\mathrm{PT}})(va)$,
    return $get\_descr_i(mem, pa)$
Else **return** $fault$

The hypervisor not only reads the PT, but also modifies them. We thus define virtual setters for the descriptors in Definition 3.2.12. Similarly to the getter, we need a physical setter to express it, we call it set_descr_i:

$$set\_descr_i : Mem \times Addr \times Descr_i \to Mem + Oob$$

We do not detail its definition as it is similar to the getter.

Definition 3.2.12 (Virtual Set Descriptor)**.** $\forall mem \in Mem$, $\forall base_{\mathrm{PT}} \in Addr$, $\forall \boldsymbol{va} \in Addr$ and $descr_i \in Descr$:
$\boldsymbol{set\_v\_descr_i(mem, base_{\mathrm{PT}}, va, descr_i)} =$
If $pt(mem, base_{\mathrm{PT}})(\boldsymbol{va}) \notin Fault$
Let $pa \in Addr$ s.t.$(pa, \_) = pt(mem, base_{\mathrm{PT}})(\boldsymbol{va})$,
return $set\_descr_i(mem, pa, descr_i)$
Else **return** *fault*

3.2.3 Set of Addresses Mapped by a Page Table

As mentioned in the previous section, an address whose image by a PT is in *Fault* is *not mapped*, otherwise it is *mapped* with some rights. For a page table $table \in PT$, we write $Im(table)$ for the intersection of the codomain of $table$ with $Addr \times Rights$, and we denote the first projection of $Im(table)$ by $Map(table)$.

Definition 3.2.13 (Mapped with RW rights)**.** Let $table \in PT$, we denote by $Map_{\mathrm{RW}}(table)$ the set of all the physical addresses mapped with Read/Write (RW) user rights by $table$:

$$Map_{\mathrm{RW}}(table) = \{pa | (pa, rw) \in Im(table)\}$$

Definition 3.2.14 (Mapped with RO rights)**.** Let $table \in PT$, we denote by $Map_{\mathrm{RO}}(table)$ the set of all the physical addresses mapped with Read Only (RO) user rights by $table$:

$$Map_{\mathrm{RO}}(table) = \{pa | (pa, ro) \in Im(table)\}$$

Definition 3.2.15 (Mapped with user rights)**.** Let $table \in PT$, we denote by $Map_{\mathrm{USR}}(table)$ the set of all the physical addresses mapped with user rights by $table$:

$$Map_{\mathrm{USR}}(table) = Map_{\mathrm{RO}}(table) \cup Map_{\mathrm{RW}}(table)$$

When reasoning about PTs, we often need to know that the memory space where SPT are stored is not mapped with user rights by any guest SPT.

Definition 3.2.16. We denote by $no\_auto\_map(mem, base)$ the property stating that the PT at address $base$ does not map itself with user rights in the memory mem.

3.3 Static Structures

Before presenting the state of the system in the next section, we present some constant of the system.

3.3.1 Memory Layout

The memory layout is static. Figure 3.2 depicts a memory layout for two guests. Each region corresponds to a particular set of permissions.

FIGURE 3.2: Example of Physical Memory Layout for two Guests

Static Permissions

An active guest is identified by an index of type Idx. Static permissions define the access rights for each guest:

Definition 3.3.1 (Static Permissions Type)**.**

$$Perm = \begin{cases} priv : Idx \to \{Addr\} \\ shared : Idx \times Idx \to \{Addr\} \\ pool : Idx \to \{Addr\} \end{cases}$$

The $priv$ field defines, for each guest, the set of addresses that the guest is allowed to map with RW rights, and that no other guest is allowed to map. The $shared$ field defines, for each couple of guest, the set of addresses that can be mapped in RW by the former and in RO by the latter (write buffer from the former to the latter). The *pool* field defines, for each guest, the addresses where its SPTs are kept. The guest is not allowed to map any address of the *pool*.

We work with static permissions that we denote by $perm \in Perm$, they do not change during the whole execution and verify the following property:

Property 3.3.1 (Disjoint Spaces)**.** The sets of addresses defined by $perm$ are disjoint.

Hypervisor Space

We define $hypspace$, a set of addresses that a guest cannot access. It corresponds to a part of the memory where the hypervisor stores its structures, in particular the Host Page Tables (HPT).

Property 3.3.2 (No Permissions for Hypervisor Space)**.** The sets of addresses defined by $perm$ are disjoints from the set of addresses defined by $hypspace$:

$$\forall i, j \in Idx, \forall a \in Addr, \begin{cases} a \in perm.priv(i) \\ \vee a \in perm.shared(i,j) \\ \vee a \in perm.pool(i) \end{cases} \Rightarrow a \notin hypspace$$

3.3.2 Host Page Table

As described in Section 1.1.3 and illustrated in Figure 1.3, the HPT are needed in order to resolve a page fault. We introduce here the pointer to the HPTs, which does not change over execution. We refer to this pointer as $base_{\mathrm{HPT}}$.

3.4 Low-Level State of the Hypervisor

As illustrated in Figure 2.1, the low-level state St of the system can be decomposed into two components. First, the state of the hardware, of type St_{HW}, which stores the actual

values of the registers, and the state of the memory. Secondly the state of the hypervisor, of type St_{HYP}, which stores the virtualized values of the hardware components presented to each guest, i.e. the virtual machines on which the guests execute. σ_{HW} and σ_{HYP} are independent. In particular, as explained in Section 3.4.1, we do not represent in memory the state of the hypervisor, therefore a modification of the hypervisor state has no effect on the hardware memory, and the other way around.

Definition 3.4.1 (System State)**.**

$$St = \begin{cases} \sigma_{\mathrm{HW}} : St_{\mathrm{HW}} & \textit{(State of the hardware)} \\ \sigma_{\mathrm{HYP}} : St_{\mathrm{HYP}} & \textit{(State of the hypervisor)} \end{cases}$$

We present the state of the hardware in Section 3.4.1, and the state of the hypervisor in Section 3.4.2.

3.4.1 Hardware State

We model the state of the hardware with the following type:

Definition 3.4.2 (Hardware State)**.**

$$St_{\mathrm{HW}} = \begin{cases} mem : Mem & \textit{(Memory)} \\ mode : Mode & \textit{(Processor mode)} \\ base_{\mathrm{PT}} : Addr & \textit{(Pointer to the current PT)} \\ regs_{\mathrm{mmu}} : Regs_{\mathrm{mmu}} & \textit{(Coprocessor 15 registers)} \\ regs_{\mathrm{core}} : Regs_{\mathrm{core}} & \textit{(Core registers)} \\ apsr : Apsr & \textit{(Application Program Status Register)} \\ regs_{\mathrm{gic}} : Regs_{\mathrm{gic}} & \textit{(GIC registers)} \end{cases}$$

Each field captures the state of a particular hardware component, we give further details below.

Memory

The field $mem \in Mem$ captures the state of a delimited part of memory. We do not take the hypervisor memory space into account, except for the PTs about which we want to reason. Indeed, each time the hypervisor performs an action, it has an impact on its memory (e.g. pushing/popping something on its stack), and reasoning about these side-effects while reasoning about the effects of the action is not conceivable.

Moreover, we do not model devices, and the hypervisor does not accommodate Direct Memory Access (DMA). DMA is a mechanism that allow devices to communicate with memory directly, bypassing the CPU, thus saving time. I/O MMU [Iom] (or SMMU [MN11; Smm]) is a hardware component which allows the hypervisor to control the access to memory by a PT mechanism similar to the MMU. Figure 3.3 illustrates the configuration: the PTs between the bus and the devices are present only when the I/O MMU or SMMU is used. Without such extensions, DMA-aware devices can access directly the memory, without any control by the PT. As any part of the memory can be accessed, it is impossible to establish isolation within any model.

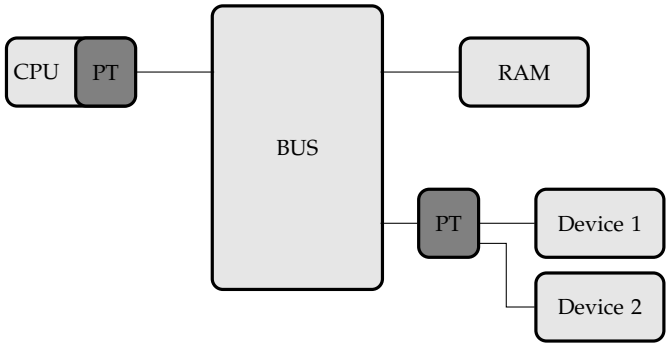


FIGURE 3.3: Direct Memory Access with I/O MMU or SMMU

Modes

In the ARMv7 design that we use, 6 execution modes are defined. Five of them are *privileged* modes, they are at the Privileged Level 1 (PL1). One mode, the *user mode*, is unprivileged. It is at Privileged Level 0 (PL0).

The field $mode$ captures the five bits of the ARM Current Program Status Register (CPSR) which indicate the current execution mode.

Guests run in user mode. It means that they can only access a restricted set of registers. Intuitively, they cannot access any register that changes the current setup, or which could extend their rights. For example, an entity running in the user mode cannot modify the $mode$ field.

The hypervisor only uses the unprivileged user (USR) mode and four of the privileged modes:

- The abort mode (*ABT*) is entered each time the system tries to access an address which is not mapped by the current Page Table.
- The supervisor mode (*SVC*) is entered when a guest makes a *hypercall*. More precisely, we do not use virtualization support, therefore there is no distinction between supervisor call and hypercall. When an application running on top of a guest makes a supervisor call, the hypervisor intercepts the call and inject it to the guest, so that it can handle it.
- The *UND* mode is entered when a guest attempts to do something which is undefined.
- The interrupt request mode (*IRQ*) is entered when an interrupt is pending and not masked.

The hypervisor does not handle the Fast Interrupt Request (FIQ) mode. This mode is similar to the IRQ mode, but is faster. As there is no handler for this mode, the hypervisor would crash if the mode were entered. However, the hypervisor configures the interrupt handling such that the FIQ mode is never entered, and guests cannot change the configuration (see Section 3.4.1). Therefore the system cannot be in this mode.

Definition 3.4.3 (Mode Type)**.** The set of modes $Mode$ gathers the user mode, and the five privileged modes : supervisor, abort, undefined, IRQ and FIQ:

$$Mode = \{usr, svc, abt, und, irq, fiq\}$$

Application Program Status Register

The CPSR mentioned previously also contains bits that can be read and written in user mode, they give information about the instruction computation. It is called the *Application* Program Status Register (APSR).

As for the other bits of the CPSR, they are masks bits, unchanged by user, and they are always set to the same value before the guest transition, so we do not model them here for clarity's sake, but we model them in our implementation.

Core Registers

The record $regs_{\text{core}}$ represents the values of the thirteen *general purpose registers*, and of the three *special purpose registers*, namely the stack pointer, the link register and the program counter. The stack pointer indicates the top of the stack, the program counter indicates the instructions to execute, while the link register holds the address to return to when a function call completes. The core register type is given in Definition 3.4.4

Definition 3.4.4 (Core Registers)**.**

$$Regs_{\text{core}} = \begin{cases} r_0 & : Reg \\ \dots & \\ r_{12} & : Reg \\ sp & : Reg \\ lr & : Reg \\ pc & : Reg \end{cases}$$

The core registers are the unprivileged registers. As we will develop in the presentation of the guest transition (Section 3.5.1), these registers are used and modified by the guest while it executes. Note that we do not model their state while the hypervisor executes, because we cannot reason both on the C code executed by the hypervisor and on the corresponding assembly instructions. We do not model the *banked* registers, because they are not modified directly by the guest, and, just as for the core registers, we do not update them when the hypervisor is executing. We only model their virtualized version (see Section 3.4.2).

The $apsr$ and the $regs_{\text{core}}$ are the unprivileged registers, they can be read and written in user mode.

Coprocessor 15

The coprocessor 15 is the Memory Management Unit (MMU). The Translation Table Base Register 0 (TTBR0) is the register of the coprocessor 15 which holds the physical address of the PTs currently used to translate virtual to physical addresses. We model it by the $base_{\text{PT}}$ register.

The field $regs_{\text{mmu}}$ holds the other registers of interest of the coprocessor 15.

Definition 3.4.5 (MMU Registers)**.**

$$Regs_{\text{mmu}} = \begin{cases} fsr & : Reg \\ far & : Addr \end{cases}$$

The Fault Address Register (FAR) and the Fault Status Register (FSR) hold the necessary information to handle a page fault. The FAR holds the virtual address which

triggered the fault, whereas the FSR holds complementary information, such as whether the access was made on a read or a write.

Generic Interrupt Controller

The Generic Interrupt Controller (GIC) holds registers needed for the interrupt handling. It is the first to be notified of an interrupt. More specifically, when an interrupt is triggered by the hardware, if the interrupt signaling is enabled in the GIC, and if it is not masked (some bits in the CPSR), the GIC marks the IRQ as pending and raises an IRQ or a FIQ exception, depending on the configuration.

The registers of the GIC are not all "real" registers, they are stored in memory. However, the modification of the GIC does not impact any region of memory defined by the permissions, and reciprocally (see Definition 3.3.1 for permissions). Indeed, the region of memory where the registers are stored do not correspond to any region to which the guest has some permissions or in which SPTs are stored. As we prove that a guest may only modify directly the regions to which it has permissions, it means that the guest cannot directly modify the configuration of the GIC.

Furthermore, the hypervisor itself does not modify the configuration of the GIC, or when it does, it put it back to its initial configuration before restoring the guest. For example, when the hypervisor masks some interrupts, it unmasks them before restoring the guest execution. Therefore, we consider that the registers related to the configuration of the GIC are constants, and they are not included in $regs_{\text{gic}}$. $regs_{\text{gic}}$ only contains the information related to which interrupt is pending. As the GIC has no impact on the memory management, we do not model these registers in detail.

Caches

It is important to mention that we do not model caches. In particular, we do not model the Translation Lookaside Buffer (TLB) (explained in Section 1.1.3).

Reasoning about the TLB involves concurrency. Indeed, the updates of the TLB are performed concurrently to the CPU execution. The execution of some code might thus hit a page fault while the MMU is updating the TLB.

As for any cache, the hardware looks first in the TLB for a mapping, and if not present, looks in the PTs. An issue arises with the use of a TLB when the mappings in the TLB are not in synchronization with the mappings of the current PT. As we are interested by isolation properties, we are only concerned if a guest can take advantage of the bad synchronization of the PTs with the TLB. Yet firstly, the TLB is flushed when the hypervisor switches guests. Secondly, as every translation present in the TLB of a guest was present at some point in the SPT of that guest, and as the definition of our allowed regions of memory is static, we can safely say that the use of the TLB does not alter our properties.

3.4.2 Hypervisor State

The hypervisor state St_{HYP}, whose type is defined in Definition 3.4.6, keeps the index of the current guest, the list of guests to schedule, and the states of all the guests.

Definition 3.4.6 (Hypervisor State)**.**

$$St_{\text{HYP}} = \begin{cases} curr : Idx & \textit{(Index of the current guest)} \\ runqueue : list < Idx > & \textit{(List of all the active guests)} \\ vcpus : Idx \to vCPU & \textit{(State of all the active guests)} \end{cases}$$

The guest state holds the data needed to provide a virtual hardware interface to the guest:

Definition 3.4.7 (Guest State)**.**

$$vCPU = \begin{cases} base_{\mathrm{SPT}} : Addr & \textit{(Pointer to the SPT)} \\ frange : \{Addr\} & \textit{(SPT virtual addresses reserved for hypervisor)} \\ alloc : \{Addr_{pg}\} & \textit{(Free pages for SPT space)} \\ phys2virt : Addr \rightarrow Addr & \textit{(Function for translation from phys to virt)} \\ regs_{\mathrm{virt}} : Regs_{\mathrm{virt}} & \textit{(Virtual registers of the Guest)} \end{cases}$$

The $base_{\mathrm{SPT}}$ contains the pointer to the SPT currently used by the guest. In particular, when a guest executes, the hypervisor puts the $base_{\mathrm{SPT}}$ of that guest in the hardware $base_{\mathrm{PT}}$ register. In the C code that we model, the hypervisor can store several SPT for one guest, it thus avoids to flush the whole SPT when a guest switches GPT. We have taken that into account in our concrete model and in our proofs on the concrete model. However, for conciseness, we only present here a state where the hypervisor maintains one SPT for each guest. Indeed, considering only one SPT alleviates the expression of all the properties presented in Chapter 4, and it is not essential for our isolation property. The abstract model that we present corresponds to a model where only one SPTs per guest is used. We discuss the impact on the abstract model of modeling one or several SPT in Chapter 5.

The $frange$, $alloc$ fields, and $phys2virt$ function concern the $pool$, i.e. the place in memory where the SPTs of a guest are stored. In our implementation of the concrete model, we normally have a field in the guest state which represents the $pool$. We have an invariant stating that the $pool$ corresponds to the $pool$ defined in the static permissions (Section 3.3.1). As the proof of this invariant is trivial, we have removed this field from the model presented here and we only define the pool in the static permission. It allows to alleviate the model. We still define the $frange$, $alloc$ and $phys2virt$ fields related to it.

The $frange$ field models the forbidden range for that guest. It is a set of guest virtual addresses (GVA) in the SPT that the guest is not allowed to access. More precisely, when an exception occurs, the execution jumps to the virtual address defined in the corresponding exception vector. If this address is not mapped in the SPT, the hypervisor would crash. Virtual addresses in $frange$ are therefore used to map exception handlers in the SPT, this is why no mapping of the forbidden range should be modified by the guest.

The $alloc$ field is the set of free page addresses located in the pool. The hypervisor uses these free pages to allocate new PTs.

The $phys2virt$ function is the reciprocal function of the SPT and HPT on the addresses of the *pool* region. As we will detail in Section 4.1.2, the hypervisor needs to translate physical addresses to virtual addresses in order to go through the SPTs.

The virtual registers $regs_{\mathrm{virt}}$ defined in Definition 3.4.8 model the emulated registers for the guest.

Definition 3.4.8 (Virtual Registers)**.**

$$Regs_{\mathrm{virt}} = \begin{cases} vcpsr : Reg \\ vregs_{\mathrm{core}} : vRegs_{\mathrm{core}} \\ vregs_{\mathrm{bnk}} : vRegs_{\mathrm{bnk}} \\ vregs_{\mathrm{mmu}} : vRegs_{\mathrm{mmu}} \\ vregs_{\mathrm{gic}} : vRegs_{\mathrm{gic}} \end{cases}$$

We define below each field of the virtual registers.

Virtual Mode

For the guest, we model the whole CPSR. The emulated mode, held into the $cpsr$ field, provides the mode in which the guest is supposed to be, and permits to load the correct banked registers before restoring the guest. The hypervisor does not handle the Fast Interrupt Request Mode (FIQ), but virtualizes it for the guest.

Virtual Core and Banked Registers

The $vregs_{\text{core}}$ emulate the registers r_0 to r_{12} plus the pc. When a guest stops its execution, the hardware r_0 to r_{12} are stored into its $vregs_{\text{core}}$, and the hardware lr is stored into the emulated pc. In this particular case, lr contains the *exception* return address.

Definition 3.4.9 (Virtual Core Registers)**.**

$$vRegs_{\text{core}} = \begin{cases} r_0 & : Reg \\ \dots & \\ r_{12} & : Reg \\ pc & : Reg \end{cases}$$

Contrarily to the general purpose registers which are shared between all the execution modes, the *banked* registers belong to only one mode.

All the modes have their own instance of sp and lr registers. In addition, all privileged modes have a banked Application Program Status Register (SPSR) (Definitions 3.4.10 and 3.4.11). The banked SPSR of a mode m holds the value of the CPSR which was active before entering m. It is used to restore the previous state. The FIQ mode also bank some general purpose registers to speed up the context switch (Definition 3.4.12).

Definition 3.4.10 (User Banked Registers)**.**

$$Bnk_{\text{PL0}} = \begin{cases} sp : & Reg \\ lr : & Reg \end{cases}$$

Definition 3.4.11 (Privileged Banked Registers)**.**

$$Bnk_{\text{PL1}} = \begin{cases} sp : & Reg \\ lr : & Reg \\ spsr : & Reg \end{cases}$$

Definition 3.4.12 (FIQ Banked Registers)**.**

$$Bnk_{\text{fiq}} = \begin{cases} r_8 : & Reg \\ \dots & \\ r_{12} : & Reg \\ sp : & Reg \\ lr : & Reg \\ spsr : & Reg \end{cases}$$

The virtual banked registers hold the banked registers of each mode (Defintion 3.4.13).

Definition 3.4.13 (Virtual Banked Registers)**.**

$$Regs_{\text{bnk}} = \begin{cases} bnk_{\text{usr}} & : Bnk_{\text{PL0}} \\ bnk_{\text{svc}} & : Bnk_{\text{PL1}} \\ bnk_{\text{abt}} & : Bnk_{\text{PL1}} \\ bnk_{\text{und}} & : Bnk_{\text{PL1}} \\ bnk_{\text{irq}} & : Bnk_{\text{PL1}} \\ bnk_{\text{fiq}} & : Bnk_{\text{fiq}} \end{cases}$$

The banked registers are used to facilitate mode switching. We describe their use in the inject transitions, in Section 3.5.3.

MMU Registers

The emulated PT base pointer ($base_{\text{GPT}}$) of the guest contains a pointer to the GPT, i.e. the PTs that maps GVA to IPA. When a page fault occurs on a GVA, the hypervisor uses the $base_{\text{GPT}}$ to find the corresponding GPA. Note that $base_{\text{GPT}}$ is an IPA, i.e. it is considered as a physical address by the guest, but is in fact virtualized by the HPT.

Similarly to the hardware state, we model the fault status and the fault address registers. We also model the flag of the SCTLR register which indicates if the MMU is activated. Indeed, the guest might not use the PTs, in this case, the SPT only contains a translation from GPA to PA, instead of GVA to PA (see figure 1.3). For conciseness, we do not model the other bits of the SCTLR, but we have in the implementation.

We denote by $vregs_{\text{mmu}}$ the emulated registers of the MMU:

Definition 3.4.14 (Virtual MMU Registers)**.**

$$vRegs_{\text{mmu}} = \begin{cases} base_{\text{GPT}} : Addr & \textit{(IPA pointing to GPT)} \\ paging : Bool & \textit{(Is MMU activated)} \\ fsr : Reg & \textit{(Fault Status Register)} \\ far : Reg & \textit{(Fault Address Register)} \end{cases}$$

Generic Interruption Controller Registers

As for the hardware state, we do not model the registers related to interruptions in details.

3.5 Low-Level Transitions

Now that we have defined the system state, we define the transitions. A transition is a relation between two states: $\rightarrow : St \times St$. We decompose a transition in four sub-transitions, of the same type. The flow of execution is shown in Figure 3.4, and described as follows:

- Step $0 \rightarrow 1$: the *hypervisor* restores the execution of the guest, in particular, it sets the processor to the unprivileged *user mode*.
- Step $1 \rightarrow 2$: the *guest* executes until it raises an exception or makes a hypercall, making the hardware switch to a privileged mode of execution.
- Step $2 \rightarrow 3$: the *hypervisor* saves the registers in the dedicated virtualized register structures.
- Step $3 \rightarrow 0$: the *hypervisor* handles the call or the fault.

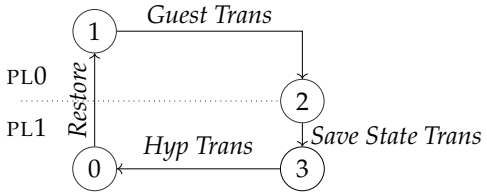


FIGURE 3.4: A Transition of the Concrete System

We describe each type of sub-transition below, and we give their formal definition. In the transitions described thereafter:

- we let $st \in St$, $\sigma_{\text{HW}} \in St_{\text{HW}}$ and $\sigma_{\text{HYP}} \in St_{\text{HYP}}$ be such that:

$$\boldsymbol{st} = \langle \boldsymbol{\sigma}_{\text{HW}}, \boldsymbol{\sigma}_{\text{HYP}} \rangle$$

- we introduce $mem \in Mem$, $mode \in Mode$, $base \in Addr$, $regs_{\text{core}} \in Regs_{\text{core}}$, $regs_{\text{mmu}} \in Regs_{\text{mmu}}$ and $regs_{\text{gic}} \in Regs_{\text{gic}}$, such that:

$$\boldsymbol{\sigma}_{\text{HW}} = \langle \boldsymbol{mem}, \boldsymbol{mode}, \boldsymbol{base}, \boldsymbol{regs}_{\text{core}}, \boldsymbol{regs}_{\text{mmu}}, \boldsymbol{regs}_{\text{gic}} \rangle$$

- we let $i \in Idx$, $runqueue \in list < Idx >$, $vcpus \in (Idx \to vCPU)\rangle$, such that:

$$\boldsymbol{\sigma}_{\text{HYP}} = \langle \boldsymbol{i}, \boldsymbol{runqueue}, \boldsymbol{vcpus} \rangle$$

For readability, fields modified by a transition are represented in boldface.

3.5.1 Guest Transition

A guest transitions occurs in *user mode*. We confine the possible effects it can have on the system by using the three properties provided by the hardware specification, namely Properties 3.5.1, 3.5.2 and 3.5.3.

Property 3.5.1 (Writable Registers)**.** An execution in user mode may only change the hardware non-privileged registers ($regs_{\text{core}}$ and $apsr$), and $regs_{\text{mmu}}$. In particular, it *cannot* change the PT base register. An interrupt may occur, modifying $regs_{\text{gic}}$. The guest transition ends with an exception or a hypercall, modifying the $mode$ field.

Property 3.5.2 (Writable Memory)**.** An execution in user mode may only modify the memory mapped with user RW rights by the PT currently used by the hardware.

Property 3.5.3 (Accessible Memory)**.** An execution in user mode only depends on the part of the memory which is mapped with user rights in the PT currently used by the hardware, and on the user mappings defined by this PT.

These two last properties are exploitable only if the so-called current PT is constant during a guest transition, i.e. if the current PT verifies the no_auto_map property (cf Definition 3.2.16). This property is ensured by an invariant stating that the part of the memory space where SPTs are stored (i.e. the $pool$) is not mapped in RW by any guest.

We give the definition of the guest transition in Definition 3.5.1. Notice that the transition depends on some extra variable o. This o represents an external oracle. An interrupt

may happen anytime while the guest is running, modifying the $regs_{\text{gic}}$. An interrupt also has an indirect impact on memory and registers. Indeed, when an interrupt occurs, the GIC, depending on its configuration, may raise an IRQ exception, thus stopping the guest execution. We have not sufficient information to decide whether an interrupt is raised or not, thus we use an external oracle. Our oracle o is to be interpreted as a list of interrupt predictions. Each time the CPU performs an atomic action on behalf of the guest (moves a value from a register to a RAM or vice-versa, performs an operation between two values, compares two values...), the oracle is read to decide if an interrupt is raised. As mentioned in Section 3.4.1, the configuration of the GIC is static, therefore the treatment of a hardware interrupt does not depends on the state of the GIC, and consequently, neither does the guest transition. This is captured by Axiom 3.5.2.

Definition 3.5.1 (Guest Transition)**.** If the system is in an unprivileged state, and if the PT currently used by the hardware does not map itself with user RW rights, if the forbidden range of the current guest is not mapped with user rights by the current PT, then a guest transition is defined.

If $no\_auto\_map(mem, base)$
$st.\sigma_{\text{HYP}}.vcpus(i).frange \notin Map_{\text{USR}}(pt(st.\sigma_{\text{HW}}.mem, st.\sigma_{\text{HW}}.base_{\text{PT}}))$

Then $\langle \boldsymbol{mem}, base, \boldsymbol{pl0}, \boldsymbol{regs_{core}}, \boldsymbol{apsr}, \boldsymbol{regs_{mmu}}, \boldsymbol{regs_{gic}} \rangle, \sigma_{\text{HYP}} \xrightarrow{GuestTrans(o)}$
$\langle \boldsymbol{mem'}, base, \boldsymbol{pl1}, \boldsymbol{regs'_{core}}, \boldsymbol{apsr'}, \boldsymbol{regs'_{mmu}}, \boldsymbol{regs'_{gic}} \rangle, \sigma_{\text{HYP}}$

Properties 3.5.1 to 3.5.3 are specifications of ARMv7 design, we report them in our model through axioms and characteristics of the guest transition. More specifically, the Property 3.5.1 is captured by the definition of the guest transition. The two other properties on the hardware (Properties 3.5.2 and 3.5.3) are expressed by the Axioms 3.5.1 and 3.5.2 in our model. Axiom 3.5.2 also states that the transition of the guest does not depend on the $regs_{\text{mmu}}$ and $regs_{\text{gic}}$. We have given a justification for the $regs_{\text{gic}}$. As for the $regs_{\text{mmu}}$, it only contain the fsr and far, which are written on a page fault, but the guest has no access to them, therefore the guest transition cannot depend on them.

Axiom 3.5.1. (Writable Memory) If the byte at a physical address pa in memory is modified after a guest transition, then pa is mapped with user RW rights by the PT currently in the hardware.

Let $st \in St$
$pa \in Addr$

If $st \xrightarrow{GuestTrans(o)} st'$
$st.\sigma_{\text{HW}}.mem(pa) \neq st'.\sigma_{\text{HW}}.mem(pa)$

Then $pa \in Map_{\text{RW}}(pt(st.\sigma_{\text{HW}}.mem, st.\sigma_{\text{HW}}.base_{\text{PT}}))$

In order to define Axiom 3.5.2, we define the two properties, $same\_map$ and $same\_user\_regs\_hw$.

Definition 3.5.2 (Same Map)**.** The property $same\_map$ holds for two memories and two PT base addresses iff the two PTs define the same user mappings and if the two memories are equals on the physical addresses mapped with user rights by the PTs.

$\forall mem_1, mem_2 \in St$, let $A = Map_{\text{USR}}(pt(mem_1, base_1))$,

$same\_map(mem_1, mem_2, base_1, base_2) \Leftrightarrow$

$$
\begin{array}{l}
Map_{\mathrm{USR}}(pt(mem_2, base_2)) = A, \\
mem_1 \overset{A}{=} mem_2, \\
\forall va \in Addr, pt(mem_1, base_1)(va) = (pa, r) \wedge r \in \{rw, ro\} \Rightarrow \\
pt(mem_2, base_2)(va) = (pa, r)
\end{array}
$$

Definition 3.5.3 (Same User Registers in Hardware)**.** Two states have the same non-privileged registers if they have the same core registers and APSR.

$$
\begin{array}{l}
\forall st_1, st_2 \in St, \\
\quad same\_user\_regs\_hw(st_1, st_2) \Leftrightarrow \quad st_1.\sigma_{\mathrm{HW}}.regs_{\mathrm{core}} = st_2.\sigma_{\mathrm{HW}}.regs_{\mathrm{core}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad st_1.\sigma_{\mathrm{HW}}.apsr = st_2.\sigma_{\mathrm{HW}}.apsr
\end{array}
$$

Axiom 3.5.2. (Accessible Memory) If the user mappings defined by the PTs of two states are equals, if the memory of these two states are equal on the physical addresses mapped with user rights by their PTs, and if the two states have the same non-privileged registers, then the guest transition modifies equally the two states.

$$
\begin{array}{ll}
\text{Let} & st_1, st_2 \in St, \\
\text{If} & same\_map(st_1.mem, st_2.mem, st_1.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}}, st_2.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}}) \\
 & same\_user\_regs\_hw(st_1, st_2) \\
 & st_1 \xrightarrow{GuestTrans(o)} st_1' \\
\text{Then} & st_2 \xrightarrow{GuestTrans(o)} st_2' \\
 & same\_map(st_1'.mem, st_2'.mem, st_1'.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}}, st_2'.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}}) \\
 & same\_user\_regs\_hw(st_1', st_2')
\end{array}
$$

The guest transition (Definition 3.5.1) and the axioms on the hardware behavior (Axioms 3.5.1 and 3.5.2) formally define our attacker model.

From the definition of the guest transition (Definition 3.5.1) and Axiom 3.5.1, we trivially obtain Theorem 3.5.1.

Theorem 3.5.1. (Unchanged PT) The PT currently used by the hardware is not modified during the guest transition.

$$
\begin{array}{ll}
\text{Let} & st \in St \\
\text{If} & st \xrightarrow{GuestTrans(o)} st' \\
\text{Then} & st'.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}} = st.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}} \\
 & pt(st'.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}}) = pt(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}})
\end{array}
$$

3.5.2 Save State Transition

Saving the state of a guest is the first action done by the hypervisor after a privileged mode is entered. We separate it from the hypervisor transitions presented in the next section for the sake of clarity. Indeed, even if this transition is performed by the hypervisor, we make it commute with the abstract guest transition. More precisely, we will see in Chapter 5 that the abstract guest transition commutes with the concatenation of the load state transition, the concrete guest transition and the save state transition.

Furthermore, as we have explained previously, the guest transition ends when an exception is raised. The hardware automatically switches to a privileged mode and jumps to the address indicated in the exception vector (modulo an offset). The exception vector indeed holds the address for the handler. The exception vectors and the handlers should be mapped ($mapped\_handlers$). If not, their would be a page fault when trying to access

these addresses. It would raise another exception and so on, making the hypervisor inoperative. Therefore, the save state transition is defined only if $mapped\_handlers$ holds. We assume that it always holds and we explain in Section 4.2.1 why this is an acceptable assumption. However *we do prove* that the guest cannot access the physical memory where the handlers are (see Section 4.2.1). This is important from a security point of view, because if guests were able to modify the handlers, their could be a privilege escalation.

The function that saves the state of a guest copies the core registers into the virtual core registers and dedicated banked registers, depending on the current emulated mode of the guest:

$$save\_state : vCPU \times Regs_{\text{core}} \to vCPU$$

This function only modify the fields $regs_{\text{virt}}.regs_{\text{core}}$ and $regs_{\text{virt}}.regs_{\text{bnk}}$ of the guest, and the APSR bits of $regs_{\text{virt}}.vcpsr$.

The save state sub-transition that foregoes every hypervisor transition is the following:

Definition 3.5.4 (Save State)**.** The registers are saved into the guest dedicated structures.

Let $\quad \boldsymbol{vcpus'} = vcpus[i \leftarrow save\_state(vcpus(i), regs_{\text{core}})]$

If $\quad mapped\_handlers(mem, vcpus)$

Then $\quad \sigma_{\text{HW}}, \langle i, runqueue, vcpus\rangle \xrightarrow{SaveState}$
$\sigma_{\text{HW}}, \langle i, runqueue, \boldsymbol{vcpus'}\rangle$

Reciprocally, the function $load\_state$ loads the emulated registers of a guest into the hardware:

$$load\_state : vCPU \to Regs_{\text{core}} \times Apsr$$

It is part of the restore transition. The hypervisor indeed loads the registers of the guest into the hardware before restoring it. It does the contrary as $save\_state$ does. In particular, it only depends on the fields $regs_{\text{virt}}.regs_{\text{core}}$, $regs_{\text{virt}}.regs_{\text{bnk}}$ and of the mode and APSR bits of the CPSR of the guest it is restoring.

3.5.3 Hypervisor Transitions

Hypervisor transitions happen in a privileged mode. The first thing done by the hypervisor is to save the state of the current guest in the guest state structure. As this operation is common to all the hypervisor transitions, and to avoid writing it in hypervisor transition, we present it separately in Section 3.5.2. The fourteen hypervisor transitions presented below happen after saving the state of the current guest.

We divide the fourteen hypervisor transitions in four groups, as illustrated in the Figure 3.5. We will see in Chapter 5 that one group of concrete transition is abstracted into one abstract transition.

The first group contains the six transitions related to the memory management: they either modify the current SPTs or the base pointer. Intuitively, they modify the guest's representation of memory.

The second group only contains the scheduling transition. It corresponds to the guest context switch, that loads the registers of the new guest. In particular, it changes the PT base pointer.

The third group contains three transitions that inject a fault to the guest and the emulation of access to privileged registers. An injection means that the hypervisor emulates the change of mode in the guest. This group also contains the access to different privileged registers. The hypervisor either writes to some emulated privileged register or

Concrete Transition	Group
Page Fault without MMU Page Fault with MMU Flush Flush All Switch Enable/Disable MMU	Memory Management
Schedule	Schedule
Inject SWI Inject UND Inject ABT Access Privileged Registers	Modify Registers
Handle IRQ Passthrough IRQ Fetch IRQ	GIC

FIGURE 3.5: Hypervisor Transitions

reads its value and puts it in one of the core register. The transitions of this group have impact on virtual registers.

Finally, the transitions of the fifth group concern the IRQs, they have an impact on the GIC registers of the hypervisor or of the guests.

Memory Management Transitions

Memory management transitions modify the SPTs of a guest. They do not modify the memory of the guest, in the sense that they do not modify the bytes in the memory accessible by the guest. Rather, they change the accessibility to certain memory locations of a guest (or several guests).

Page Fault A *page fault* occurs when the guest tries to access an address which is not mapped in its current SPT.

When the guest does not activate MMU, it does not use its GPT. Yet the hypervisor still maintains SPT in order to control access to memory. In this case, the SPT of a guest are a partial copy of the HPT, with lower rights. The transition presented in Definition 3.5.6 shows the adding of a new mapping in the SPT after a page fault, when MMU is disabled.

When MMU is activated, the hypervisor go through the GPT in order to find out to which IPA the faulting virtual address corresponds. If no translation for the faulting address exists in the GPT, then the hypervisor notifies the guest that the translation for the address is missing. To do so it injects the fault in the guest, we detail the injection transitions in Section 3.5.3.

If a translation is present in the GPT of the guest, the hypervisor composes the GPT with the HPT. It then maps the virtual address to the physical address found by this composition in the SPT. It does not map just the address, but the whole page containing this address, to a whole physical page. This mapping is added by the map function, we give its formal definition in Chapter 4. The transition modifies the *pool* of that guest (hence the modified memory field), and the state of the guest, when it needs to allocate a new page (Definition 3.5.7).

In the transitions defined in Definitions 3.5.7 and 3.5.6, a mapping is added to the SPT only if the rights requested are allowed. We define the $allowed$ function (Definition 3.5.5).

Definition 3.5.5 (Allowed)**.** Let $i \in Idx$ and $pa \in Addr$ a physical address:

$$allowed(pa, i, rw) \Leftrightarrow \exists j \in Idx, pa \in shared(i, j) \vee pa \in perm.priv(i)$$

$$allowed(pa, i, ro) \Leftrightarrow \exists j \in Idx, pa \in shared(j, i) \vee allowed(pa, i, rw)$$

Definition 3.5.6 (Page Fault Without MMU)**.** If the fault was triggered by guest accessing some address ipa [1]; if the current guest has MMU disabled [2]; if ipa is mapped in the HPT to some PA pa [3] with suitable rights [4]; then the Page Fault without MMU transition is defined and consists in mapping the address ipa to the physical address pa in the current SPT:

If
[1] $decode(\sigma'_{\text{HW}}) = pf(\boldsymbol{ipa})$
[2] $vcpus(i).regs_{\text{virt}}.vregs_{\text{mmu}}.paging = false$
[3] $pt(mem, base_{\text{HPT}})(\boldsymbol{ipa}) = (\boldsymbol{pa}, \boldsymbol{r}_0)$
[4] $allowed(\boldsymbol{pa}, i, r_0)$
[5] $pt(mem', base_{\text{SPT}}{}') = pt(mem, base_{\text{SPT}})[\boldsymbol{ipa} \leftarrow (\boldsymbol{pa}, \boldsymbol{r}_0)]$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{PageFault}$
$\langle \boldsymbol{mem'}, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus'}\rangle$

Definition 3.5.7 (Page Fault With MMU)**.** If the fault was triggered by guest accessing some address gva [1]; if the current guest has MMU enabled [2]; if gva is mapped in the GPT [3,4] to some IPA ipa with suitable rights [5]; if this ipa translates into some pa in the HPT; and if the map operation succeeds[7]; then the Page Fault with MMU transition is defined and consists in mapping the virtual page at gva to the physical page at pa in the current SPT:

If
[1] $decode(\sigma'_{\text{HW}}) = pf(\boldsymbol{gva})$
[2] $vcpus(i).regs_{\text{virt}}.vregs_{\text{mmu}}.paging = true$
[3] $pt(mem, base_{\text{HPT}})(vcpus(i).base_{\text{GPT}}) = (pbase_{\text{GPT}}, \_)$
[4] $pt(mem, pbase_{\text{GPT}})(\boldsymbol{gva}) = (ipa, \boldsymbol{r}_0)$
[5] $allowed(\boldsymbol{pa}, i, r_0)$
[6] $pt(mem, base_{\text{HPT}})(ipa) = (\boldsymbol{pa}, \_)$
[7] $(\boldsymbol{mem'}, \boldsymbol{vcpus'}) = map(mem, vcpus, i, \boldsymbol{gva}, \boldsymbol{pa}, \boldsymbol{r}_0)$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{PageFault}$
$\langle \boldsymbol{mem'}, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus'}\rangle$

Flush When the guest has MMU activated, it maintains the translation cache, i.e. the TLB. When it unmaps one or several pages in its GPT, it sends a *flush* or *flush all* instruction to flush the cache. The hypervisor intercepts these operations and uses it to synchronize the SPT with the GPT.

Similarly as for the page fault operation, we use $unmap$ and $unmap\_all$ functions in the following definitions but we detail them in Chapter 4.

The *flush one* operation removes one entry in the second level PT, i.e. it removes a page. It does not return a second level PT to the pool, even when it removes the last entry in the second level PT. This transition only has impact on memory (Definition 3.5.8).

The *flush all* operation removes all the entry in the current SPT, except the entries of the forbidden range. It returns the second level PT to the pool. Thus this transition not only has impact on memory, but also on the state of the current guest, more specifically on its *free* field (Definition 3.5.9).

Definition 3.5.8 (Flush One Entry)**.** If the guest called flush on gva [1]; if the current guest has MMU enabled [2]; if gva is not in the forbidden range of the current guest [3]; and if the unmap operation succeeds[4]; then the Flush One Entry transition is defined and consists in unmapping the virtual page containing gva in the current SPT:

If [1] $decode(\sigma'_{\text{HW}}) = \mathit{flush}(\boldsymbol{gva})$
[2] $vcpus(i).regs_{\text{virt}}.vregs_{\text{mmu}}.paging = true$
[3] $\boldsymbol{gva} \notin vcpus(i).frange$
[4] $\boldsymbol{mem'} = unmap(mem, vcpus, i, \boldsymbol{gva})$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{FlushOne}$
$\langle \boldsymbol{mem'}, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle$

Definition 3.5.9 (Flush All)**.** If the guest called flush all [1]; if the current guest has MMU enabled [2]; and if the $unmap\_all$ operation succeeds[3]; then the Flush All transition is defined and consists in unmapping all the addresses outside the forbidden range in the current SPT:

If [1] $decode(\sigma'_{\text{HW}}) = \mathit{flush\_all}$
[2] $vcpus(i).regs_{\text{virt}}.vregs_{\text{mmu}}.paging = true$
[3] $(\boldsymbol{mem'}, \boldsymbol{vcpus'}) = unmap\_all(mem, vcpus, i)$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{FlushAll}$
$\langle \boldsymbol{mem'}, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus'}\rangle$

Switch The switch transition happens when the current guest switches its current PT, i.e. when it schedules a new process. As mentioned in Section 3.4.2, the hypervisor only stores one SPT for each guest. The switch transition performs two changes: it flushes the whole SPT, and it changes the current GPT base address.

Definition 3.5.10 (Switch)**.** If the guest called switch all [1]; if the current guest has MMU enabled [2]; and if the $unmap\_all$ operation succeeds[3]; then the Switch transition is defined and consists in unmapping all the addresses outside the forbidden range in the current SPT and changing the guest emulated GPT base pointer:

If [1] $decode(\sigma'_{\text{HW}}) = switch(base_{\text{GPT}new})$
[2] $vcpus(i).regs_{\text{virt}}.vregs_{\text{mmu}}.paging = true$
[3] $(\boldsymbol{mem'}, \boldsymbol{vcpus'}) = unmap\_all(mem, vcpus, i)$
[4] $\boldsymbol{vcpus''} = vcpus'[(i).regs_{\text{virt}}.vregs_{\text{mmu}}.base_{\text{GPT}} \leftarrow base_{\text{GPT}new}]$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{Switch}$
$\langle \boldsymbol{mem'}, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus''}\rangle$

Enable/Disable MMU The guest might enable (resp. disable) its MMU. The hypervisor flushes the current SPT and sets the emulated guest register controling MMU to enabled (resp. disabled).

Definition 3.5.11 (Enable/Disable MMU)**.** If the guest enables/disables its MMU [1]; if the $unmap\_all$ operation succeeds [2]; then the Enable/Disable MMU transition is defined and consists in unmapping all the addresses outside the forbidden range in the current SPT and changing the guest register controlling MMU activation:

If [1] $decode(\sigma'_{\text{HW}}) = mmu(bool)$
[2] $(\boldsymbol{mem'}, \boldsymbol{vcpus'}) = unmap\_all(mem, vcpus, i)$
[3] $\boldsymbol{vcpus''} = vcpus'[(i).regs_{\text{virt}}.vregs_{\text{mmu}}.paging \leftarrow bool]$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{e/d\ MMU}$
$\langle \boldsymbol{mem'}, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus''}\rangle$

Schedule Transition

The guest may send a hypercall to suspend its execution, in this case the hypervisor schedules another guest. It stores the registers of the current guest and loads the registers of the next one.

Definition 3.5.12 (Schedule)**.** If a scheduling is triggered and if the runqueue is not empty, then the Scheduling transition is defined and consists in loading the new current guest MMU settings in the hardware:

If $decode(\sigma'_{\text{HW}}) = sched$
$runqueue$ not empty

Let $< j :: runqueue' >= runqueue$
$runqueue_{new} = runqueue' :: i$
$base' = vcpus(j).base_{\text{SPT}}$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{Sched}$
$\langle \boldsymbol{mem'}, mode, \boldsymbol{base'}, \boldsymbol{regs'_{\text{core}}}, \boldsymbol{apsr'}, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle \boldsymbol{j}, \boldsymbol{runqueue}_{new}, \boldsymbol{vcpus'}\rangle$

GIC Transitions

In case of an interrupt request which is not pushed to the guest, the hypervisor handles the IRQ (Definition 3.5.13). The change due to the IRQ is captured by the $regs_{\text{gic}}$ field, it does not affect any other component of the state.

Some IRQs are not handled by the hypervisor, they are forwarded to the guests. They are pushed into the IRQ queue of some or all the guests. They might change the state of all the guests, as described in Definition 3.5.14.

The guest can fetch an IRQ through the page fault handler, when the guest accesses external devices (Definition 3.5.15).

We do not give details about the functions used in the definitions below (push/pop an IRQ), because they are obvious and will not be needed afterwards.

Definition 3.5.13. (Handle IRQ) If there is an IRQ to handle, the handle IRQ transition is defined and consists in the hypervisor handling the IRQ, without interfering with guest's state:

If $decode(\sigma'_{\text{HW}}) = irq\_handler$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{handleIRQ}$
$\langle mem, mode, base, regs'_{\text{core}}, apsr, regs_{\text{mmu}}, \boldsymbol{regs'_{\text{gic}}}\rangle, \langle i, runqueue, vcpus\rangle$

Definition 3.5.14. (Passthrough IRQ) If a passthrough IRQ is triggered, the hypervisor forwards the IRQ to the guests:

Let $\boldsymbol{vcpus'}$ s.t. $\forall j, vcpus'(j) = push(vcpus(j).regs_{\text{gic}}, n)$

If $decode(\sigma_{\text{HW}}) = passthrough\_irq(n)$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{passthroughIRQ}$
$\langle mem, mode, base, regs'_{\text{core}}, apsr, regs_{\text{mmu}}, \boldsymbol{regs'_{\text{gic}}}\rangle, \langle i, runqueue, \boldsymbol{vcpus'}\rangle$

Definition 3.5.15. (Fetch IRQ) The guest pop an IRQ:

Let $\boldsymbol{vcpus'} = vcpus[i.regs_{\text{virt}}.regs_{\text{gic}} \leftarrow pop(i.regs_{\text{virt}}.regs_{\text{gic}}, n)]$

If $decode(\sigma'_{\text{HW}}) = fetch\_irq(n)$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}} \rangle, \langle i, runqueue, \boldsymbol{vcpus} \rangle \xrightarrow{fetchIRQ}$
$\langle mem, mode, base, regs'_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}} \rangle, \langle i, runqueue, \boldsymbol{vcpus'} \rangle$

Modify Registers Transitions

Each mode has some banked registers which indicate where to return (banked lr) and in which mode (banked $mode$). When the guest sends a hypercall to switch to X mode, the hypervisor emulates the way ARMv7 handles the change of mode with banked registers:

1. The pc and the emulated $cpsr$ are stored respectively in the emulated banked lr and $spsr$ of mode X.
2. The mode of the emulated $cpsr$ is changed into X mode.
3. pc is set to the exception vector corresponding to X mode.

The inject function defined below performs these steps.

Definition 3.5.16. (Inject) $\forall vcpu \in vCPU$, $X \in pl1$, we define the function $inject$ as follow:

Let $vregs_{\text{core}} = vcpu.regs_{\text{virt}}.vregs_{\text{core}}$
$regs'_{\text{bnk}} = regs_{\text{bnk}}[bnk_X.spsr \leftarrow regs_{\text{virt}}.cpsr, bnk_X.lr \leftarrow vregs_{\text{core}}.pc]$
$vregs'_{\text{core}} = vregs_{\text{core}}[pc \leftarrow X\_vec,]$
$cpsr' = set\_mode(vcpu.regs_{\text{virt}}.cpsr, X)$
$regs'_{\text{virt}} = vcpu.regs_{\text{virt}}.[cpsr \leftarrow cpsr', vregs_{\text{core}} \leftarrow vregs'_{\text{core}}, regs_{\text{bnk}} \leftarrow regs'_{\text{bnk}}]$

Then $inject(vcpu, X) = vcpu[regs_{\text{virt}} \leftarrow regs'_{\text{virt}}]$

We give the definition of inject SVC and inject UND in Definition 3.5.18. The inject ABT occurs in a different context. It corresponds to the case where the hypervisor forwards the page fault to the guest, so that it adds a new mapping in its GPT that the hypervisor would shadow.

Definition 3.5.17 (Inject SVC/UND Transition)**.** If the guest attempts to switch to mode $X \in \{svc, und\}$, the inject transition is defined as follows:

Let $\boldsymbol{vcpus'} = vcpus[i \leftarrow inject(vcpus(i), X)]$

If $decode(\sigma'_{\text{HW}}) = inject(X)$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}} \rangle, \langle i, runqueue, vcpus \rangle \xrightarrow{InjectX}$
$\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}} \rangle, \langle i, runqueue, \boldsymbol{vcpus'} \rangle$

Definition 3.5.18 (Inject ABT Transition)**.** If the fault is triggered by an access to some address gva [1]; if the current guest has MMU enabled [2]; if gva is not mapped in the GPT [3,4]; then the Inject ABT transition is defined and consists in injecting the fault into the guest:

Let $vcpus' = vcpus[i \leftarrow inject(vcpus(i), X)]$
$\boldsymbol{vcpus''} = vcpus[i.regs_{\text{virt}}.vregs_{\text{mmu}}.fsr \leftarrow regs_{\text{mmu}}.fsr,$
$i.regs_{\text{virt}}.vregs_{\text{mmu}}.far \leftarrow regs_{\text{mmu}}.far]$

If [1] $decode(\sigma'_{\text{HW}}) = pf(\boldsymbol{gva})$
[2] $vcpus(i).regs_{\text{virt}}.vregs_{\text{mmu}}.paging = true$
[3] $pt(mem, base_{\text{HPT}})(vcpus(i).base_{\text{GPT}}) = (pbase_{\text{GPT}}, \_)$
[4] $pt(mem, pbase_{\text{GPT}})(\boldsymbol{gva}) \in Fault$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{InjectABT}$
$\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus''}\rangle$

Finally, the guest might want to read or write to privileged registers. A write would modify one of the privileged registers emulated in $regs_{\text{virt}}$. A read would change some emulated core register (the hypervisor reads the value and write it to an unprivileged register).

Definition 3.5.19 (Modify Registers)**.** If the guest attempts to access some privileged registers, the hypevisor accesses the emulated registers on its behalf:

Let $\boldsymbol{vcpus'} \cong vcpus[i.regs_{\text{virt}}]$

If $decode(\sigma'_{\text{HW}}) = inject(X)$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{ModifyRegs}$
$\langle mem, mode, base, regs'_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus'}\rangle$

3.5.4 Restore Transition

Finally, after it finishes handling a fault or a guest request, the hypervisor restores the state of the guest, so that it can execute (guest transition).

Before restoring the guest, the hypervisor might inject an IRQ to the guest. For clarity's sake, we divide the restore transition in two sub-transitions:

1. Inject an IRQ or do nothing.
2. Load the state of the guest to run.

Definition 3.5.20 shows the inject IRQ transition. If there is no pending IRQ, then this transition is a nop, and the hypervisor transition is directly followed by the load state transition of Definition 3.5.21.

Definition 3.5.20 (Inject IRQ Transition)**.** If the system is in a privileged mode, and if an interrupt is pending and not mask, then the IRQ is injected

Let $\boldsymbol{vcpus'} = vcpus[i \leftarrow inject(vcpus(i), irq)]$

If $pending\_irq(vcpus(i).regs_{\text{gic}})$
$\neg mask(vcpus(i).regs_{\text{gic}})$

Then $\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{InjectIRQ}$
$\langle mem, mode, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, \boldsymbol{vcpus'}\rangle$

Definition 3.5.21 (Load State Transition)**.** If the system is in a privileged mode, then the state of the guest is loaded and hardware mode is set to usr.

Let $(\boldsymbol{regs'_{\text{core}}}, \boldsymbol{apsr'}) = load\_state(vcpus(i))$

If $mode \in pl_1$

Then $\langle mem, \boldsymbol{mode}, base, regs_{\text{core}}, apsr, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle \xrightarrow{Restore}$
$\langle mem, \boldsymbol{usr}, base, \boldsymbol{regs'_{\text{core}}}, \boldsymbol{apsr}, regs_{\text{mmu}}, regs_{\text{gic}}\rangle, \langle i, runqueue, vcpus\rangle$

3.6 Key Points

- The permissions of each guest to some physical memory areas are defined statically.
- The concrete state is composed of the hardware state and the hypervisor state.
- In the hypervisor state, the hypervisor stores data to emulate the hardware for each guest. In particular, for each guest, it stores the pointer to the SPT and data necessary to the management of the SPT (forbidden addresses, allocator, location in virtual memory).
- We work with two level PTs.
- Our concrete system has fourteen transitions. Five of them concern memory management. It means that they have effect on the physical addresses accessible to a guest.

We have presented the concrete transition system. In the next chapter, we will present the invariant properties of the model. We will show their preservation over to most critical memory management transitions.

Chapter 4

Invariant Properties of the System

Contents

The correspondence between transitions of the concrete and abstract system, that we will present in Chapter 5, can only be established on certain conditions. Indeed the abstraction rely on the specification of the effects of each transitions, which can be specify only if the concrete state verifies some properties.

For example, consider the Figure 4.1, which depicts the mapping of a virtual address whose indexes in the first and second level PT are idx_1 and idx_2, in the PT at address $pbase_1$. Suppose that $pbase_1$ is a pointer to the SPT of some guest. When the hypervisor maps a new page in this SPT, it may allocate memory in order to store a second level PT (when the descriptor at idx_1 is a fault).

1. If the allocation is bugged, for example if the newly allocated second level PT is not empty, the effect of the map operation is that the guest gains access to several new pages of addresses which were not intended to be mapped.

2. If a property ensures that the free PTs available for allocation are empty, the observable effects are as expected: the guest gains access only to addresses located in the page expressly mapped. We prove that such properties are *invariants* of the system.

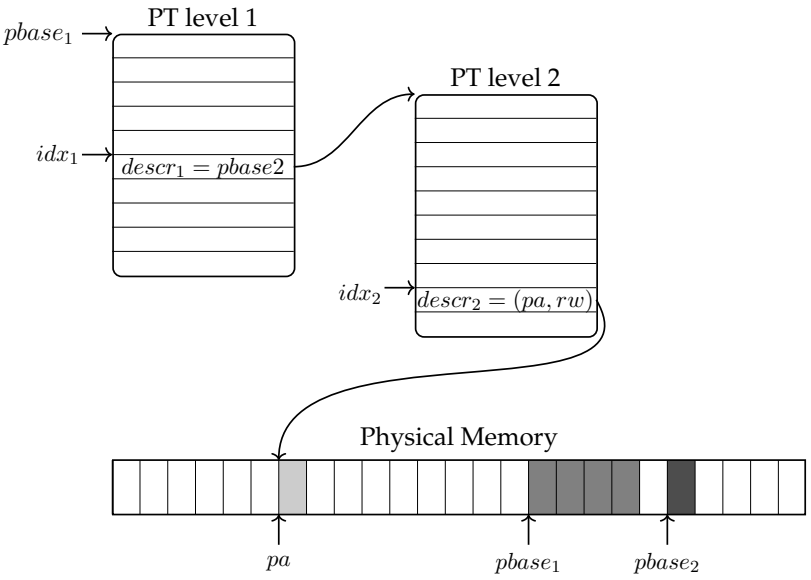


FIGURE 4.1: Mapping from a virtual address $va = idx_1 \oplus idx_2$ to pa, in a PT located at $pbase_1$.

The invariants hold between every transitions, in particular at the beginning of each transition where they are needed. Some of them are only required for preserving other invariants over a particular transition, this is the case of the property we have just mentioned. Others are needed for almost every transitions, this is the case of invariants on PTs presented in Section 4.1.1.

We divide our invariants in three groups, the PT well-formedness invariants and the translation of the pool space invariants are properties on the PTs, they are presented in Section 4.1. The third group gathers the invariants specifics to some transitions and are presented in Section 4.2. We list them informally below:

- PT well-formedness:
  - The first level HPT is included in the hypervisor space.
  - The second level HPTs are included in the hypervisor space.
  - The first level SPT of a guest does not overlap with its second level SPTs.
  - The second level SPTs of a guest does not overlap with its second level SPTs.
  - The first level SPT of a guest is in its pool.
  - The second level SPTs of a guest are in its pool.
- Translation of pool space:
  - Specification of how the pool of a guest is mapped in the HPT.
  - Specification of how the pool of a guest is mapped in its SPTs.
- Invariants specifics to some transitions:
  - Physical addresses mapped in the SPT of a guest are allowed for that guest.
  - The pool of a guest is mapped by virtual addresses in the forbidden range of the guest's SPTs.

- A page listed as free is not mapped.
- A page not listed as free is mapped.
- The forbidden range is not mapped with user rights.

A concrete state verifying all the invariants is called a *well-formed* state. We give the definition of a well-formed state at the end of this chapter (Definition 4.4.1). The proofs of refinement in Chapter 5 are stated on well-formed states.

In this chapter we define all these invariants, we analyze their inter-dependencies and justify why each invariant is needed. In Section 4.1 we present the invariants concerning the PTs and the properties that can be deduced. In Section 4.2 we present invariants that are required for some specific operations. Sections 4.3.4, 4.2.2, 4.2.3 and 4.3.3 analyze in details some transitions described in Chapter 3: we enunciate the specification properties needed in order to abstract them, and we show on which invariants their proof rely.

4.1 Invariants on Page Tables

PT invariants are at the base of almost all the properties we have proved. Invariants of Section 4.1.1 show that the SPT base pointers stored in a state effectively points to well formed PT structures, i.e. that they can be interpreted as function from virtual to physical addresses. Invariants of Section 4.1.2 specify how the SPTs are themselves mapped, and thus allow to translate back physical addresses to virtual addresses for this particular set of addresses.

4.1.1 Page Tables Well-formedness

The properties on PTs are related to their location in memory. As illustrated in Figure 4.1, PT are themselves stored in the physical memory they map. We want to make sure that no PT overlap with another, and that they are all located in safe parts of the memory, i.e. not accessible in user mode.

Note that, for readability's sake, we do not present here the reasoning about intervals. Indeed, an interval $[a, a+b[$ makes sense only if $a < a+b$, which might not be the case if $a+b$ overflows. This is something we take into account in our proof, but do not present here.

In order to define the invariants in this section, we often need to express whether a second level PT is a component of a PT. The property of Definition 4.1.1 specifies the belonging to a PT for second level PT. Note that the base address of a PT is the base address of its first level PT, therefore we do not need to define a similar predicate for first level PT.

Definition 4.1.1. (Reachable Base 2) . The base address $base_2$ of a second level PT is reachable from the base $base$ of a PT through idx_1 if the idx_1^{th} descriptor of the first level PT at $base$ contains a link to $base_2$:

$\forall mem \in Mem,$
$\forall base, base_2 \in Addr,$
$\forall idx_1 \in Idx,$

$$reachable_{base_2}(mem, base, idx_1, base_2) \Leftrightarrow$$
$$idx_1 < 4096 \wedge get\_descr_1(fetch(mem, compute_1(base, idx_1)) = base_2$$

Definitions 4.1.1 and 4.1.2 define invariants ensuring that the first and second level PTs of HPT are in the hypervisor space. As all the modifications we perform on memory concern the $pools$ or the memory regions of the guest, these invariants are required to prove that the HPTs are never tampered with.

Definition of Invariant 4.1.1 (HPT First Level in Hypervisor Space)**.** The property $hpt\_pt1\_hypspace$ is verified iff the first level PT of the HPT is located within the hypervisor space:

$$hpt\_pt1\_hypspace \Leftrightarrow [base_{\mathrm{HPT}} + size_{PT1}[\in hypspace$$

Definition of Invariant 4.1.2. (HPT Second Levels in Hypervisor Space) The property $hpt\_pt2\_hypspace$ is verified iff all the second level PTs of the HPT are located within the kernel memory:

$\forall mem \in Mem,$

$$\begin{aligned} hpt\_pt2\_hypspace(mem) \Leftrightarrow \quad & (\forall base_2 \in Addr, \forall idx_1 \in Uint, \\ & reachable_{base_2}(mem, base_{\mathrm{HPT}}, idx_1, base_2) \Rightarrow \\ & [base_2 + size_{PT2}[\subset hypspace) \end{aligned}$$

Invariants of Definitions 4.1.3 and 4.1.4 ensure that the PT of level 1 and 2 of the SPT of each guest are in the $pool$ of that guest. Again, it is used to isolate the SPTs from the other parts of the memory, in particular from user space, $hypspace$, and from other $pools$.

Definition of Invariant 4.1.3. (SPT First Level in Pool) The property $spt1\_in\_pool$ is verified iff the first level PT of the SPT of a guest is located in its pool.

$\forall vcpus \in (Idx \rightarrow vCPU),$

$$\begin{aligned} & spt1\_in\_pool(vcpus) \Leftrightarrow \\ & \quad \forall i \in Idx, [vcpus(i).base_{\mathrm{SPT}}, vcpus(i).base_{\mathrm{SPT}} + size_{PT1}[\subset perm(i).pool \end{aligned}$$

Definition of Invariant 4.1.4. (SPT Second Levels in Pool) The property $spt1\_in\_pool$ is verified iff the second level PTs of the SPT of a guest is located in its pool.

$\forall mem \in Mem,$
$\forall vcpus \in (Idx \rightarrow vCPU),$

$$\begin{aligned} spt2\_in\_pool(mem, vcpus) \Leftrightarrow \quad & (\forall i \in Idx, \forall base_2 \in Addr, \forall idx_1 \in Uint \\ & reachable_{base_2}(mem, vcpus(i).base_{\mathrm{SPT}}, idx_1, base_2) \Rightarrow \\ & [base_2 + size_{PT2}[\subset perm(i).pool) \end{aligned}$$

Invariants of Definitions 4.1.5 and 4.1.6 ensure that, within a same PT, first and second level PTs do not overlap with each other.

Definition of Invariant 4.1.5. (SPT First Level no overlap with Second Levels) The property $no\_overlap\_pt1$ is verified iff the first level PT of the SPT of a guest does not overlap with any of its second level PT.

$\forall mem \in Mem,$
$\forall vcpus \in Idx \rightarrow vCPU,$

$$\begin{aligned} & no\_overlap\_pt1(mem, vcpus) \Leftrightarrow \\ & \quad (\forall i \in Idx, \forall base_2 \in Addr, \forall idx_1 \in Uint \\ & \quad reachable_{base_2}(mem, vcpus(i).base_{\mathrm{SPT}}, base_2) \Rightarrow \\ & \quad [vcpus(i).base_{\mathrm{SPT}}, vcpus(i).base_{\mathrm{SPT}} + size_{PT1}[\cap [base_2 + size_{PT2}[= \{\}) \end{aligned}$$

Definition of Invariant 4.1.6. (SPT Second Levels no overlap with Second Levels) The property $no\_overlap\_pt2$ is verified iff the second level PTs of the SPT of a guest does not

overlap with any of its other second level PT.

$\forall mem \in Mem,$
$\forall vcpus \in (Idx \to vCPU),$

$$no\_overlap\_pt2(mem, vcpus) \Leftrightarrow \begin{array}{l}(\forall i \in Idx, \forall base_2, base'_2 \in Addr, \forall idx_1, idx'_1 \in Uint \\ reachable_{base_2}(mem, vcpus(i).base_{\mathrm{SPT}}, idx_1, base_2) \wedge \\ reachable_{base_2}(mem, vcpus(i).base_{\mathrm{SPT}}, idx'_1, base'_2) \wedge \\ idx_1 \neq idx'_1 \Rightarrow \\ [base_2 + size_{PT2}[\cap [base'_2 + size_{PT2}[= \{\})\end{array}$$

With these six invariants, we are able to reason on the effects of *physical* setters on memory. Consider a second level descriptor which is *safe* to add in a SPT of a guest. For example, a pointer to a page inside a *private* region of a guest is a safe second level descriptor. If we set a second level entry in the SPT of a guest to a so called safe descriptor, then:

1. No mapping is modified in the HPT, because of Invariants 4.1.1, 4.1.2, 4.1.3 and 4.1.4, which isolate the HPT from the SPTs.
2. No mapping is modified in the SPT of another guest, because of invariants 4.1.3 and 4.1.4, which isolate the SPT of one guest from the others.
3. No other modification than the one intended is performed in the SPT, because of invariants 4.1.5 and 4.1.6.

4.1.2 Translation of Hypervisor Virtual Space

As we mentioned in Section 3.2.2, memory can only be accessed with virtual addresses, so when the hypervisor needs to access some physical address, it uses *phys2virt* to translate it to a virtual address before accessing it. This function is defined for each guest in its state (field *phys2virt* in Definition 3.4.7).

The function *phys2virt* performs a mere addition. It defines how the HPT and SPT of a guest map the addresses of a pool. In particular, in our case, a contiguous segment of virtual memory is mapped contiguously in physical memory. This function could be more complicated as long as it is static, as we use *phys2virt* for both HPT and SPT. For each *phys2virt*, we define *virt2phys*, the inverse function.

The properties *hpt_phys2virt* and *spt_phys2virt* defined below state that the addresses of the pool are mapped with *phys2virt* translation by the HPT and SPT.

Definition of Invariant 4.1.7. (HPT Mapped Translation) The addresses of the SPT region of one guest are mapped by a translation in the HPT.

$\forall mem \in Mem,$
$\forall vcpus \in (Idx \to vCPU),$
Let $virt2phys$ be the inverse function of $vcpus(i).phys2virt$,

$$hpt\_phys2virt(mem, vcpus) \Leftrightarrow \begin{array}{l}(\forall i \in Idx, \forall \boldsymbol{va} \in Addr, \\ virt2phys(\boldsymbol{va}) \in perm.pool(i)) \Rightarrow \\ pt(mem, base_{\mathrm{HPT}})(\boldsymbol{va}) = (virt2phys(\boldsymbol{va}), pl_1))\end{array}$$

Definition of Invariant 4.1.8. (SPT Mapped Translation) The addresses of the SPT region of one guest are mapped by a translation in this SPT.

$\forall mem \in Mem,$
$\forall vcpus \in (Idx \to vCPU),$

$$spt\_phys2virt(mem, vcpus) \Leftrightarrow \begin{array}{l}(\forall i \in Idx, \forall \boldsymbol{va} \in Addr,\\ virt2phys(\boldsymbol{va}) \in perm.pool(i)) \Rightarrow\\ pt(mem, vcpus(i).base_{\mathrm{SPT}})(\boldsymbol{va}) = (virt2phys(\boldsymbol{va}), pl_1))\end{array}$$

Virtual to Physical Getters/Setters The invariants of Section 4.1.1 allow to specify the effects of the modification of a SPT with a physical setter (set_descr_i). However, as explained in Section 3.2.2, the hypervisor modifies the SPT using virtual addresses, i.e. using $set\_v\_descr_i$ (Definition 3.2.12). The invariants of this section allow to complete the properties provided by the invariants of the previous section: they allow to reason about a modification of descriptors accessed by *virtual* addresses.

When the hypervisor accesses the idx^{th} entry of the SPT at base $base_{\mathrm{SPT}}$, it first translates the base address to a virtual address, then computes the virtual address of the descriptor from this address, and finally accesses the descriptor at this virtual address with the virtual getter, as follows:

1. $vbase_{\mathrm{SPT}} = phys2virt(base_{\mathrm{SPT}})$.
2. $vpde = compute(vbase_{\mathrm{SPT}}, \boldsymbol{idx})$.
3. $get\_v\_descr_1(mem, base_{\mathrm{HPT}}, vpde)$ (or $get\_v\_descr_1(mem, base_{\mathrm{SPT}}, vpde)$, depending which PT is currently in use).

We prove that the three steps are actually equivalent to the simpler steps involving only physical accesses:

1. $ppde = compute(base_{\mathrm{SPT}}, \boldsymbol{idx})$.
2. $get\_descr_1(mem, ppde)$

This result, for virtualization with SPT, is summed up in Lemma 4.1.2. We have a second lemma stating this result for virtualization with HPT.

The result stated in Lemma 4.1.1 is used in the proof of Lemma 4.1.2. It is directly obtained from the definition of $compute$ (Definition 3.2.9), since the function $virt2phys$ is linear.

Lemma 4.1.1 (Compute commutes). $\forall i \in Idx, \forall vbase_{\mathrm{SPT}} \in Addr_{pg}, \forall idx_1 \in Uint$,
Let $virt2phys = st.\sigma_{\mathrm{HYP}}.guests(i).virt2phys$

$$virt2phys(compute(vbase_{\mathrm{SPT}}, idx)) = compute(virt2phys(vbase_{\mathrm{SPT}}), idx)$$

Lemma 4.1.2. (SPT Get Phys to Virt) The virtual access to an idx^{th} descriptor in a PT can be related to the physical access the idx^{th} descriptor in the same PT, if the PT used for virtualization is equivalent to the $virt2phys$ function:
$\forall idx_1 \in Uint$,
$\forall mem \in Mem$,
$\forall vcpus \in (Idx \to vCPU)$,
$\forall i \in Idx$,
Let $base_{\mathrm{SPT}} = vcpus(i).base_{\mathrm{SPT}}$
[1] $vbase_{\mathrm{SPT}} = phys2virt(base_{\mathrm{SPT}})$
[2] $vpde = compute(vbase_{\mathrm{SPT}}, idx)$
[3] $ppde = compute(base_{\mathrm{SPT}}, idx)$

If $spt1\_in\_pool(mem, vcpus)$
$hpt\_phys2virt(mem, vcpus)$
$idx_1 < 4096$

Then $get\_v\_descr_1(mem, base_{\mathrm{HPT}}, vpde) = get\_descr_1(mem, ppde)$

Proof. Since $virt2phys(vpde) \in perm.pool(i)$, by Definition 4.1.8, invariant $spt\_phys2virt$ ensure that:

$$pt(mem, base_{\mathrm{SPT}})(\boldsymbol{vpde}) = (virt2phys(\boldsymbol{vpde}), \_)$$

Therefore, by Definition 3.2.11 of the virtual get descriptor:

$$get\_v\_descr_1(mem, base_{\mathrm{SPT}}, vpde) = get\_descr(mem, virt2phys(\boldsymbol{vpde}))$$

By Lemma 4.1.1, and by combining $virt2phys$ in the equality [2]:

$$virt2phys(vpde) = compute(base_{\mathrm{SPT}}, idx)$$

It means, by [3], that $virt2phys(vpde) = ppde$. Therefore:

$$get\_v\_descr_1(mem, base_{\mathrm{SPT}}, vpde) = get\_descr(mem, ppde)$$

Qed.

□

In addition, for each virtualization case, we have a similar lemma for second level descriptors, which needs $spt2\_in\_pool$ invariant to show that the address of the entry considered is in the pool, so that $hpt\_phys2virt$ (or $spt\_phys2virt$) can be applied. The proofs of these lemmas are straightforward (less than 10 hints each).

We refer to these lemmas as *equivalence of physical and virtual getter/setter*.

PT Modification Properties As explained in Section 3.2.2, the hypervisor uses virtual setters (Definition 3.2.11) when it modifies the SPTs. The virtual addresses it uses are virtualized either with HPT or with SPT, i.e. the current PT used by the MMU to translate the addresses is either the HPT or the SPT. At each modification, we verify that only the intended entry in the SPT is modified. The PT function definition uses physical addresses, so we want to compare the values returned by *physical* getters before and after the modification. So our lemmas on PT modification all consider modifying a SPT using *virtual* addresses and compare the value of some descriptors accessed before and after the modification with *physical* addresses. All in all, there are 20 lemmas:

- When setting an entry in the SPTs with HPT (resp. SPT) virtualized addresses:
  - 4 lemmas for proving that there is no effects on the HPTs (resp. SPTs).
  - 4 lemmas for proving that there is no effects on certain entries of the SPTs.
  - 2 lemmas for specifying the effect on the modified entry.

Each proof takes between 20 and 50 hints, and rely on all the properties of PTs mentioned previously. For example, let's consider the following lemma:

Lemma 4.1.3. (HPT get set first SPT) If one sets the idx_1^{th} entry in the first level PT of a SPT, using the virtual setter virtualized with HPTs, then the get of another entry stays unchanged.

$\forall mem \in Mem,$
$\forall vcpus \in (Idx \rightarrow vCPU),$
$\forall idx_1, idx'_1 \in Idx,$

Let $ppde = compute(vcpus(i).base_{\mathrm{SPT}}, \boldsymbol{idx}_1),$
$\boldsymbol{vpde} = phys2virt(ppde),$
$\boldsymbol{ppde'} = compute(vcpus(j).base_{\mathrm{SPT}}, \boldsymbol{idx}'_1),$

If $no\_overlap\_pt1(mem, vcpus),$
$spt1\_in\_pool(mem, vcpus),$
$hpt\_phys2virt(mem, vcpus),$
$\boldsymbol{mem'} = set\_v\_descr_1(\boldsymbol{mem}, base_{\mathrm{HPT}}, \boldsymbol{vpde}),$
$\boldsymbol{idx}_1 \neq \boldsymbol{idx}'_1 \vee i \neq j,$

Then $get\_descr_1(mem, \boldsymbol{ppde'}) = get\_descr_1(\boldsymbol{mem'}, \boldsymbol{ppde'})$

The property $no\_overlap\_pt1$ ensures that the modification on the first level PT does not impact first level PT of other guests. It thus provides the result when $i \neq j$. The combination of $spt1\_in\_pool$ and $hpt\_phys2virt$ allows us to specify $set\_v\_descr_1$ in terms of physical getters and setters. This way, we can actually reason on physical addresses, and ensure that a modification at idx_1 does not impact the entry at idx'_1 when $idx_1 \neq idx'_1$.

On variants of this lemma which modify the second level of PT, we need in addition the invariants on the second level PTs (such as $no\_overlap\_pt2$, $spt2\_in\_pool$).

We do not not present all the properties here, we refer to them in the remaining of the document as *PT modification properties*.

The properties discussed in this Section give an idea of the complexity induced by the PTs: the simultaneous manipulation of physical and virtual addresses, the different kind of virtualization to take into account (with HPT or SPTs), their intricate structure which multiply the risks of out of boundaries access, or overlapping. Note that these properties may seem obvious, but they highly contribute to complicate every aspect of the proof, at least every reasoning on memory access.

4.2 Invariants Specific to some Transitions

The invariants of the previous section ensure that the PTs are well formed and therefore can be interpreted as functions from virtual to physical addresses. They are the basis of abstraction. We present in this section the extra invariants we need for proving the preservation of these PT invariants over:

- the *Guest Transition*,
- the *map* operation (used in the *page fault transition*),
- the *unmap* operation (used in the *flush transition*),
- the *unmap_all* operation (used in the *flush all* and the *switch transitions*).

Each of the following four sections is dedicated to one of these operation. We give the definitions of the invariants needed for the particular operation, and we informally explain why it is needed. The map, $unmap$ and $unmap\_all$ operations have been used to define some of the MMU transitions (Section 3.5.3), but we have not defined them yet. Therefore we will begin each of the dedicated section by the definition of the operation.

Finally in Section 4.2.6, for each operation, we present the dependencies between the invariants when proving their preservation over the operation.

4.2.1 Guest Transition

When the hypervisor restores the guest, it puts the guest's own SPT in the hardware, meaning that the current PT is the SPT of the current guest. More generally, except for the map operation for which the hypervisor uses the HPTs, the addresses are always virtualized with the current guest SPTs. This is an invariant of our system. As it is temporarily broken for the operation map, we do not include it in our definition of a well-formed state, but its preservation is obviously preserved by all the transitions.

Definition 4.2.1 (SPT current PT)**.** The property spt_curr_pt holds iff the SPT base address of the current guest is equal to the PT base address.

$\forall st \in St,$

$$spt\_curr\_pt(st) \Leftrightarrow \quad st.\sigma_{\mathrm{HW}}.base_{\mathrm{PT}} = st.\sigma_{\mathrm{HYP}}.vcpus((st.\sigma_{\mathrm{HYP}}.curr).base_{\mathrm{SPT}})$$

Recall from Section 3.5.1 that the guest transition requires that the current PT in σ_{HW} does not map itself with user rights (no_auto_map, Definition 3.2.16). The property no_auto_map should then be an invariant property of the SPT of each guest.

Axiom 3.5.1 states that the guest transition only impacts parts of the memory mapped in RW by the current PT. Therefore, if we want to preserve the 8 invariants presented, we should ensure that the SPT of the current guest does not map in RW any place where the GPT or HPT are kept (i.e. the $pools$ or the $hypspace$).

These two requirements on the SPT are ensured by the following invariant:

Definition of Invariant 4.2.1. (Mapped User Allowed) The property $map\_usr\_allowed$ holds iff all the mappings present in the SPT of the guest are allowed:

$\forall mem \in Mem,$
$\forall vcpus \in (Idx \to vCPU),$

$$\begin{aligned} map\_usr\_allowed(mem, vcpus) \Leftrightarrow \quad & (\forall i \in Idx, \forall va, pa \in Addr, r \in Rights \\ & pt(mem, vcpus(i).base_{\mathrm{SPT}})(va) = (pa, r) \Rightarrow \\ & allowed(pa, i, r)) \end{aligned}$$

Indeed, the fact that an address pa is $allowed$ for guest i with some rights r (Definition 3.5.5) implies that the address is located in one of the $shared$ or $priv$ region of the guest. Yet these regions are distinct from the $pools$ and $hypspace$ (Definition 3.3.1). So the SPT of every guest does not map the $pools$ and $hypspace$, and in particular they do not map themselves. We detail these properties in Section 4.3.4.

Furthermore, the property $map\_usr\_allowed$ allow to specify the effects of the map transition, as we will see in Section 4.3.1.

Exception Handlers

We have explained in Section 3.5.2 that the guest should not be able to modify the way the exceptions are handled. When an exception is raised, the hardware switches to a privileged mode and jumps to an exception vector whose virtual address is defined in the Vector Base Address Register (VBAR). The exception vector does not contain the full code for a handler (it is only 32 bits large), it jumps to the proper handler. The handling of an exception can therefore be altered in three places, we give their access restrictions below:

1. the VBAR register is only modifiable in privileged mode,
2. the exception vector is mapped in the forbidden range,

3. the exception handler is also mapped in the forbidden range.

Invariant 4.2.2, presented below, ensures that the virtual addresses of the forbidden range only map physical addresses outside of user regions. Combined with the Invariant 4.2.1 it ensures that the guests cannot modify any mapping at a virtual address within the forbidden range and cannot access any physical address mapped by the forbidden range. Therefore the guest transition cannot modify any part of the exception handler.

Definition of Invariant 4.2.2. (Forbidden Range not User Region) The property $frange\_no\_usr$ holds iff the $frange$ are mapped to physical addresses outside of all the user regions.

$\forall mem \in Mem,$
$\forall vcpus \in (Idx \rightarrow vCPU),$

$$\begin{aligned} frange\_no\_usr(mem, vcpus) \Leftrightarrow \quad & (\forall i \in Idx, \forall va, pa \in Addr, r \in Rights, \\ & va \in frange \wedge \\ & pt(mem, vcpus(i).base_{\mathrm{SPT}})(va) = (pa, r) \Rightarrow \\ & \nexists (i, j), pa \in perm.priv(i) \vee pa \in perm.shared(i)) \end{aligned}$$

The *hypervisor* could still unmap addresses of the forbidden range, even when Invariant 4.2.2 holds. However, SPTs are only modified with one of the three functions: map, $unmap$ or $unmap\_all$. And each time these functions are called, they avoid the forbidden range (a check is made before each call to map and $unmap$ and the check is included in the function $unmap\_all$).

For the particular case of the abort handler, the hypervisor switches to HPTs to handle the fault, then switches back to SPT at the end of the handler. The HPTs are never modified after initialization, therefore the handler is never unmapped from the HPT.

Of course this is not a formal proof that the hypervisor never unmaps the handlers. However, we already have the arguments in place. Furthermore we have proved that the $pool$, which is mapped in the forbidden range, is always mapped in the HPTs and the SPTs (Invariants 4.1.7 and 4.1.8). We are therefore confident that proving that the forbidden range is never unmapped would just be a small extension. We plan to add it to our model, but for now we consider that it is reasonable to consider that the exception vectors and handlers are always mapped.

4.2.2 Map a Page

As previously explained in Section 3.5.3, when a guest tries to access a virtual address which is mapped in its GPT but not in its SPT, a page fault exception occurs and the hypervisor adds a new mapping in the SPT (Definition of the Page Fault With MMU transition 3.5.7).

The map function is precisely the function, called in the Page Fault With MMU transition, which maps the virtual address of a page to a physical address of a page in the SPT of a guest. This function is therefore called on *page* addresses, i.e. on addresses *aligned* to the size of a page.

$$map : St \times Idx \times Addr_{pg} \times Addr_{pg} \rightarrow (St + Fail)$$

The function map only modifies and depends on certain component of the state (the memory, the guests). We give its definition below.

Definition 4.2.2. (Map a Page) $\forall st \in St, \forall i \in Idx, \forall va \in Addr_{pg}$, and for all pair $(pa, r) \in Addr_{pg} \times Rights$, we define the function which maps the virtual page at address va to the

physical page at address pa with rights r in the SPT of the i^{th} guest:

Let $mem = st.\sigma_{\mathrm{HW}}.mem$,
$vcpus = st.\sigma_{\mathrm{HYP}}.vcpus$,
$phys2virt = vcpus(i).phys2virt$,
$vbase_{\mathrm{SPT}} = phys2virt(vcpus(i).base_{\mathrm{SPT}})$,
$idx_1, idx_2 \in Uint$ such that $idx_1 \oplus idx_2 = va$,
$vpde = compute(vbase_{\mathrm{SPT}}, \boldsymbol{idx}_1)$,

If $get\_v\_descr(mem, base_{\mathrm{HPT}}, vpde) \in \{Oob\}$
Then **return** *fail*

(Case: allocate a PT1 before mapping)
Elif $get\_v\_descr_1(mem, base_{\mathrm{HPT}}, vpde) \in \{Fault\}$,
Then If $vcpus(i).free$ not empty
Let $< a :: free' >= vcpus(i).free$,
$vcpus_{new} = vcpus[(i).free \leftarrow free']$,
$mem' = set\_v\_descr_1(mem, base_{\mathrm{HPT}}, vpde, a)$,
$vbase_2 = phys2virt(a)$,
$vpte = compute(vbase_2, \boldsymbol{idx}_2)$,
If $set\_v\_descr_2(mem', spt, vpte, \boldsymbol{(pa, r)}) \notin Oob$
Then Let $mem^{new} = set\_v\_descr_2(mem', base_{\mathrm{HPT}}, vpte, \boldsymbol{(pa, r)})$,
$st^{new} = st[\sigma_{\mathrm{HW}}.mem \leftarrow mem^{new}, \sigma_{\mathrm{HYP}}.vcpus \leftarrow vcpus^{new}]$,
return $\boldsymbol{st}^{new}$
Else **return** *fail*
Else **return** *fail*

(Case: PT1 is already present, add an entry in it)
Else Let $vbase_2 = phys2virt(get\_v\_descr_1(mem, base_{\mathrm{SPT}}, vpde))$,
$vpte = compute(vbase_2, \boldsymbol{idx}_2)$,
If $set\_v\_descr_2(mem, spt, vpte, \boldsymbol{(pa, r)}) \notin Oob$
Then Let $mem_{new} = set\_v\_descr_2(mem, base_{\mathrm{HPT}}, vpte, \boldsymbol{(pa, r)})$,
$st^{new} = st[\sigma_{\mathrm{HW}}.mem \leftarrow mem^{new}]$,
return $\boldsymbol{st}^{new}$
Else **return** *fail*

There are two cases for which the function is defined. In the first case (*Case: allocate a PT2 before mapping*), there is no second level PT associated to the address va to be mapped. Thus the hypervisor looks for an empty page in the *pool* of that guest, removes it from the set of free pages and set the former faulting first level descriptor to the address of this page. Then it sets the second level descriptor of this second level PT corresponding to va to the couple of physical address and right given as argument.

In the second case (*Case: PT2 is already present, add an entry in it*), there is a second level PT associated to the address va. Therefore the hypervisor only has to modify an entry in the second level PT just allocated, similarly to how it was done in the precedent case.

In the definitions that follow, we need to express the fact that a page in memory is or is not allocated to some PT:

Definition 4.2.3 (Not Allocated Page)**.** A page is not allocated to a SPT of a $vCPU$ if it does not overlap with its first level PT nor its second level PT.
$\forall mem \in Mem$,
$\forall vcpu \in vCPU$,

$$
\begin{aligned}
not\_alloc\_pg(mem, vcpu) \Leftrightarrow \quad & (\forall a \in Addr, \\
& [vcpu.base_{\mathrm{SPT}}, size_{PT1}[\cap [a, size_{page}[= \{\} \\
& \wedge \forall base_2 \in Addr, \forall idx_1 \in Uint, \\
& reachable_{base_2}(mem, vcpu.base_{\mathrm{SPT}}, idx_1, a) \Rightarrow \\
& [base_2, size_{PT1}[\cap [a, size_{page}[= \{\})
\end{aligned}
$$

When a second level PT is allocated, a whole page is allocated for the second level PT. To make sure that two PTs do not share the same second level PT, this page should not be allocated (not_alloc_pg). Furthermore, to make sure that the new PT does not already contain some pointer to other parts of memory, we ensure that every entry of the allocated PT leads to a fault, i.e. that the new second level PT is $empty$. This is ensured by Invariant 4.2.3.

Definition 4.2.4 (Empty Page)**.** The second level PT at address a is empty iff all its entries are a fault.

$\forall mem \in Mem,$
$\forall a \in Addr,$

$$
empty(mem, a) \Leftrightarrow \quad \forall idx_2 < 256, get\_descr(compute(a, idx_2)) \in Fault
$$

Definition of Invariant 4.2.3. (Free Page Empty and not Mapped) Free pages in the SPT pool of a guest are not mapped by any other guest, and are empty.

$\forall mem \in Mem,$
$\forall vcpus \in Idx \rightarrow vCPU,$

$$
\begin{aligned}
free\_not\_mapped(mem, vcpus) \Leftrightarrow \quad & (\forall i \in Idx, \forall a \in vcpus(i).free, \\
& empty(mem, a) \wedge \\
& \forall j \in Idx, not\_alloc\_pg(mem, vcpus(j)))
\end{aligned}
$$

4.2.3 Unmap a Page

The $unmap$ function removes the mapping of a virtual address va from a SPT. As the allocation and deallocation of memory addresses is made page by page, all the virtual addresses located in the same page as va are unmapped. The $unmap$ function is only called on addresses aligned to the size of a page. On an aligned address, being in the same page as va is equivalent as being in the interval $[va, va + page\_size[$.

The $unmap$ function avoids addresses which are located within the forbidden range. Indeed if the guest unmaps addresses needed for handling a fault, the hypervisor will crash.

The function $unmap$ takes a state and the index of a guest and returns a new state, or fails. As shown in the Definition 4.2.5, it only has effect on the memory field of the state. It takes as input a memory, some vCPUS and an index of vCPU and returns a new memory.

$$
unmap : St \times Idx \rightarrow St + Fail
$$

Definition 4.2.5. (Unmap a Page)$\forall st \in St, \forall i \in Idx, \forall va \in Addr_{pg}$, we define the function which unmaps the page at address va in the SPT of the i^{th} guest:

Let $mem = st.\sigma_{\mathrm{HW}}.mem$,
$vcpus = st.\sigma_{\mathrm{HYP}}.vcpus$,
$phys2virt = vcpus(i).phys2virt$,
$vbase_{\mathrm{SPT}} = phys2virt(vcpus(i).base_{\mathrm{SPT}})$,
$idx_1, idx_2 \in Uint$ such that $idx_1 \oplus idx_2 = va$,
$vpde = compute(vcpus(i).base_{\mathrm{SPT}}, idx_1)$,

If $get\_v\_descr(mem, vcpus(i).base_{\mathrm{SPT}}, vpde) \in \{Oob\}$
Then **return** *fail*

(Case: not mapped, nothing to do)
If $get\_v\_descr(mem, vcpus(i).base_{\mathrm{SPT}}, vpde) \in \{Fault\}$
Then **return** $\boldsymbol{st}$

(Case: set the idx_2^{th} entry to fault)
Else Let $vbase_2 = phys2virt(get\_v\_descr_1(mem, base_{\mathrm{SPT}}, vpde))$,
$vpte = compute(vbase_2, idx_2)$
If $set\_v\_descr_2(mem, base_{\mathrm{SPT}}, vpte, fault) \notin \{Oob\}$
Then Let $mem^{new} = set\_v\_descr_2(mem, base_{\mathrm{SPT}}, vpte, \boldsymbol{fault})$
$st^{new} = st[\sigma_{\mathrm{HW}}.mem \leftarrow mem^{new}]$
return $\boldsymbol{st}^{new}$
Else **return** *fail*

Just as the map operation, there are two cases, depending on whether a second level PT is already allocated for the address va considered. In the first case (*Case: not mapped, nothing to do*), there is no PT2 for the address va, meaning that it is already unmapped.

In the second case (*Case: set the idx_2^{th} entry to fault*), there is a second level PT, thus we set the second level descriptor corresponding to va to fault.

The $unmap$ operation only removes one entry in the second level PT. It does not deallocate the page where the second level PT stands, even when it unmaps its last entry.

Before calling the $unmap$ function, the hypervisor verifies that it does not remove an entry from the forbidden range of that guest. The following invariant states that the pool is located in the forbidden range of the guest, so that we know that the removed entry is not in a $pool$, thus preserving the PT invariants.

Definition of Invariant 4.2.4. (Pool in Forbidden Range) The property $pool\_in\_frange$ is verified iff the pool of a guest is in its forbidden range.
$\forall vcpus \in Idx \rightarrow vCPU$,

$$pool\_in\_frange(vcpus) \Leftrightarrow \forall i \in Idx, perm.pool(i) \subset vcpus(i).frange$$

4.2.4 Unmap all

The function $unmap\_all$ is called to remove all the mappings of the SPT of the current guest, except the mappings of the forbidden range. To simplify, we suppose that the forbidden range is an interval located at the end the address space, so that the function $unmap\_all$ is called on the interval from zero to the beginning of the forbidden range.

Definition 4.2.6. (Unmap All) $\forall st \in St, \forall i \in Idx$, we define the function which removes all the mapping in the SPT of the i^{th} guest until the beginning of the forbidden range:

Let $mem = st.\sigma_{\text{HW}}.mem$
$mem' = mem$
$vcpus = st.\sigma_{\text{HYP}}.vcpus$
$vcpus' = vcpus$
$phys2virt = vcpus(i).phys2virt$
$vbase_{\text{SPT}} = phys2virt(vcpus(i).base_{\text{SPT}})$
$to_1 \in Uint$ the index of the beginning of the forbidden range.

For $idx_1 = 0...(to_1 - 1)$:
  Let $vpde = compute(vcpus(i).base_{\text{SPT}}, idx_1)$
  If $get\_v\_descr(mem, vbase_{\text{SPT}}, vpde) \in \{Oob\}$
  Then **return** *fail*

  (Case: set the all the entries to fault)
  If $get\_v\_descr(vbase_{\text{SPT}}, vpde) \notin \{Fault\}$
  Then
    Let $base_2 = get\_v\_descr_1(mem, vbase_{\text{SPT}}, vpde)$
      $vbase_2 = phys2virt(base_2)$
    For $idx_2 = 0...255$
      Let $vpte = compute(vbase_2, idx_2)$
      If $set\_v\_descr_2(mem, base_{\text{SPT}}, vpte, fault) \notin \{Oob\}$
      Then $mem'' = set\_v\_descr_2(mem'', base_{\text{SPT}}, vpte, \boldsymbol{fault})$
      Else **return** *fail*
    End For
    $mem' = set\_v\_descr_1(mem', base_{\text{SPT}}, vpde, \boldsymbol{fault})$
    $free' = addsorted(vcpus(i).free, base_2)$
    $vcpus' = vcpus[(i).free \leftarrow free']$
End For
$st^{new} = st[\sigma_{\text{HW}}.mem \leftarrow mem', \sigma_{\text{HYP}}.vcpus \leftarrow vcpus']$
return $\boldsymbol{st}^{new}$

The first loop of this function go through all the entries of the first level PT of the SPT of the guest, except the entries of the forbidden range. If the entry leads to a second level PT, we enter a second loop, else we go to the next iteration. The second loop go through all the second level entries and set them to fault. Then the second level PT is returned to the *pool* and the first level descriptor is set to fault.

The invariant stating that the *pool* is in the forbidden range (Definition 4.2.4) is needed for this transition, as only the forbidden range is avoided by the unmapping.

Contrarily to the *unmap* operation, here the empty second level PTs are returned to the *pool*. In order to maintain the invariant of Definition 4.2.3, the invariant of Definition 4.2.5 must hold. So this invariant is not needed for the preservation of PT invariants, but for the preservation of a specific invariant on the *unmap_all* operation.

Definition of Invariant 4.2.5. (Not Free Page Allocated) Pages located in the SPT pool of a guest but not listed as free are mapped by this guest.

$\forall mem \in Mem$
$\forall vcpus \in Idx \rightarrow vCPU$

$$\begin{aligned} alloc\_is\_alloc(mem, vcpus) \Leftrightarrow \quad & (\forall i \in Idx, \forall a \in Addr_{pg}, \\ & a \in perm.pool(i) \wedge a \notin vcpus(i).free \Rightarrow \\ & \neg not\_alloc\_pg(mem, vcpus(i))) \end{aligned}$$

Finally, Invariant 4.2.2 is needed to prove the effects of the $unmap\_all$ operation. Indeed, we want to prove that, after a $unmap\_all$ operation, there is no more mappings with user rights in the concerned SPT. In order to prove it, we must know that the forbidden range is not mapped with user rights by the SPT of a guest (Definition 4.2.7). This definition is provided the combination of Invariants 4.2.2 and 4.2.1.

Definition 4.2.7. (Forbidden Range not User Mapped) The property $frange\_no\_map$ holds iff the $frange$ of the guests are not mapped in their SPT with user rights:
$\forall mem \in Mem,$
$\forall vcpus \in (Idx \rightarrow vCPU),$

$$\begin{aligned} frange\_no\_map(mem, vcpus) \Leftrightarrow \quad & (\forall i \in Idx, \forall va, pa \in Addr, r \in Rights, \\ & va \in frange \wedge \\ & pt(mem, vcpus(i).base_{\text{SPT}})(va) = (pa, r) \Rightarrow \\ & r = pl1) \end{aligned}$$

Even if an invariant is needed for one operation in particular, its preservation must be proved over the other transitions as well, creating a need for other invariants. We have presented here the invariants along with the transitions justifying their presence. We now show in more detail how they interact.

4.2.5 Well-formed Registers

We have an invariant ensuring that the CPSR and the SPSR stored in the virtualized banked registers of a guest have a well-formed mode. The mode is given by five bits of the CPSR (resp. SPSR), there are therefore 2^5 possible combinations, but only six of them actually correspond to a mode. The function $mode$ has the following type:

$$mode : Reg \rightarrow Mode + None$$

and may thus raise none. We present the invariant ensuring the mode registers well-formedness below:

Definition of Invariant 4.2.6 (Well-formed Mode)**.** The property wf_regs holds iff the bits of the virtual CPSR and banked SPSR of each guest encoding the mode correspond to an actual mode:
$\forall vcpus \in (Idx \rightarrow vCPU),$

$$\begin{aligned} wf\_regs(vcpus) \Leftrightarrow \quad & (\forall i \in Idx, X \in \{svc, abt, und, irq, fiq\} \\ & mode(vcpus(i).regs_{\text{virt}}.vcpsr) \notin None \wedge \\ & mode(vcpus(i).regs_{\text{virt}}.regs_{\text{bnk}}.bnk_X.spsr) \notin None) \end{aligned}$$

The virtualized CPSR and SPSRs are modified on transitions that modify registers, presented in Section 3.5.3. The hypervisor modify itself these registers and always puts one of the five defined modes. As this invariant is trivially preserved, we do not include it in the definition of a well-formed state and we do not explain its dependency with other invariants. Indeed, its preservation does not depend on the other invariants and it is not needed to prove any of the other invariants presented.

In the rest of the document, we consider that this invariant is always true, i.e. that the function $mode$ on the CPSR and SPSRs of the guests never raises none.

4.2.6 Interdependencies

We sum up the dependencies between invariants in the following tables. We separate the invariants in the three groups we have established in the previous sections: the PT invariants (Definitions 4.1.1 to 4.1.6), the specific invariants (Definitions 4.2.1 to 4.2.5) and the virtual to physical invariants (Definitions 4.1.7 to 4.1.8).

For each invariant in a line, we mark with a cross all the properties used in the proof of its preservation over the guest transition in Figure 4.2, the *map* operation in Figure 4.3, the *unmap* operation in Figure 4.4 and the *unmap_all* operation in Figure 4.5.

Proof of ↓ / Needs →	*hpt_pt1_hypspace* (4.1.1)	*hpt_pt2_hypspace* (4.1.2)	*no_overlap_pt1* (4.1.5)	*no_overlap_pt2* (4.1.6)	*spt1_in_pool* (4.1.3)	*spt2_in_pool* (4.1.4)	*map_usr_allowed* (4.2.1)	*frange_no_usr* (4.2.2)	*free_not_mapped* (4.2.3)	*pool_in_frange* (4.2.4)	*alloc_is_alloc* (4.2.5)	*hpt_phys2virt* (4.1.7)	*spt_phys2virt* (4.1.8)
(*hpt_pt1_hypspace*)	×												
(*hpt_pt2_hypspace*)	×	×			×	×	×						
(*no_overlap_pt1*)			×		×	×	×						
(*no_overlap_pt2*)				×	×	×	×						
spt1_in_pool					×								
spt2_in_pool					×	×	×						
map_usr_allowed					×	×	×						
(*frange_no_usr*)					×	×	×	×					
(*free_not_mapped*)					×	×	×		×				
(*pool_in_frange*)					×	×	×			×			
(*alloc_is_alloc*)					×	×	×				×		
(*hpt_phys2virt*)					×	×	×					×	
(*spt_phys2virt*)					×	×	×						×

FIGURE 4.2: Invariants Dependencies: for each invariant in a row, we mark with a cross all the properties used in the proof of its preservation over the ***guest*** transition.

Guest transition The table for guest transition differs significantly from the tables for the memory operations. The three invariants which are not between parenthesis, namely *spt1_in_pool*, *spt2_in_pool* and *map_usr_allowed*, allow to prove that the guest transition do not modify the memory regions where the SPTs are stored (see Lemma 4.3.7). Therefore, the preservation of all the properties on the SPTs follows easily from this lemma. Similar reasoning show that the memory region where HPT are stored is not impacted by the guest transition, thus that the transition preserves the properties on HPTs.

Map Figure 4.3, show that the preservation of every invariant, except *hpt_pt1_hypspace* (which does not depend on memory), rely on *hpt_phys2virt*. Indeed, the *map* operation

Proof of ↓ / Needs →	*hpt_pt1_hypspace* (4.1.1)	*hpt_pt2_hypspace* (4.1.2)	*no_overlap_pt1* (4.1.5)	*no_overlap_pt2* (4.1.6)	*spt1_in_pool* (4.1.3)	*spt2_in_pool* (4.1.4)	*map_usr_allowed* (4.2.1)	*frange_no_usr* (4.2.2)	*free_not_mapped* (4.2.3)	*pool_in_frange* (4.2.4)	*alloc_is_alloc* (4.2.5)	*hpt_phys2virt* (4.1.7)	*spt_phys2virt* (4.1.8)
hpt_pt1_hypspace	×												
hpt_pt2_hypspace	×	×	×		×							×	
no_overlap_pt1	×	×	×		×	×			×			×	
no_overlap_pt2	×	×	×	×	×	×			×			×	
spt1_in_pool			×		×							×	
spt2_in_pool	×	×	×	×	×	×			×			×	
(*map_usr_allowed*)	×	×	×	×	×	×	×		×			×	
(*frange_no_usr*)	×	×	×	×	×	×		×	×	×		×	
free_not_mapped	×	×	×	×	×	×			×			×	
(*pool_in_frange*)	×	×	×	×	×	×			×	×		×	
(*alloc_is_alloc*)	×	×	×	×	×	×			×		×	×	
hpt_phys2virt	×	×	×		×	×						×	
(*spt_phys2virt*)	×	×	×	×	×	×				×		×	×

FIGURE 4.3: Invariants Dependencies: for each invariant in a row, we mark with a cross all the properties used in the proof of its preservation over the ***map*** operation.

is performed with HPT virtualized addresses, therefore the *hpt_phys2virt* is needed to translate physical to virtual addresses for modification purposes. As explained in Section 4.2.2, the *free_not_mapped* invariant is required for the proof of preservation of the PT invariants.

Notice that the invariants between parenthesis are *not needed* for proof of preservation of the PT invariants, *hpt_phys2virt* and *free_not_mapped*. They are needed for other transitions. The proof of these invariants are rather independent one from each other. Except *spt_phys2virt* which relies on the *pool_in_frange* invariant.

Unmap The *unmap* operation is performed with SPT virtualized address. Therefore, compared to Figure 4.3, you can see that all the dependencies on *hpt_phys2virt* have been replaced to dependencies on *spt_phys2virt* in Figure 4.4. Furthermore, we can see that, similarly to the *map* operation, the PT invariants are mandatory for every invariant preservation. However, as *unmap* does not use the HPTs, the *hpt_pt1_hypspace* and *hpt_pt2_hypspace* invariants are not needed for any proof except for *hpt_phys2virt*. The PT invariants only rely on themselves except for *spt2_in_pool* which uses the property *pool_in_frange*. Similarly to the map operation, the invariants not between parenthesis represent the minimal set of invariants needed for the proof of their own preservation over *unmap*.

Proof of ↓ / Needs →	*hpt_pt1_hypspace* (4.1.1)	*hpt_pt2_hypspace* (4.1.2)	*no_overlap_pt1* (4.1.5)	*no_overlap_pt2* (4.1.6)	*spt1_in_pool* (4.1.3)	*spt2_in_pool* (4.1.4)	*map_usr_allowed* (4.2.1)	*frange_no_usr* (4.2.2)	*free_not_mapped* (4.2.3)	*pool_in_frange* (4.2.4)	*alloc_is_alloc* (4.2.5)	*hpt_phys2virt* (4.1.7)	*spt_phys2virt* (4.1.8)
(*hpt_pt1_hypspace*)	×												
(*hpt_pt2_hypspace*)	×	×	×	×	×	×							×
no_overlap_pt1			×		×	×							×
no_overlap_pt2			×	×	×	×							×
spt1_in_pool					×								×
spt2_in_pool			×	×	×	×				×			×
(*map_usr_allowed*)			×	×	×	×	×						×
(*frange_no_usr*)	×	×	×	×	×	×		×		×			×
(*free_not_mapped*)			×	×	×	×			×	×			×
pool_in_frange			×	×	×	×				×			×
(*alloc_is_alloc*)			×	×	×	×				×	×		×
(*hpt_phys2virt*)	×	×	×	×	×	×						×	×
spt_phys2virt			×	×	×	×				×			×

FIGURE 4.4: Invariants Dependencies: for each invariant in a row, we mark with a cross all the properties used in the proof of its preservation over the ***unmap*** operation.

Unmap All Just as for the *unmap*, the *unmap_all* operation is performed with SPT virtualized addresses. Hence all the crosses in the column *spt_phys2virt*. The way of reading this table is a bit different from the previous tables. Indeed the function *unmap_all* involves loops. We cannot prove the preservation of invariants one by one and give a fine analysis of their inter-dependencies, because we must make all the invariants depending one of another pass together as a loop invariant. We have established the proof of the invariants in three steps. First we have proved the preservation of all the PT invariants, the *pool_in_frange* invariant and the *spt_phys2virt* invariant alltogether. It means that these invariants constitute a sufficient set to prove the preservation of the PT invariants over the *unmap_all* operation.

Then, we have proved the preservation of the previous block plus the *free_not_mapped* and *alloc_is_alloc* invariants that depend one of each other. Finally we have proved the remaining invariants, which are independent from each other, on top of these block.

The granularity could be finer, for example, it should be possible to prove *hpt_pt1_hypspace* and *hpt_pt2_hypspace* independently from the other invariants.

Proof of ↓ / Needs →	*hpt_pt1_hypspace* (4.1.1)	*hpt_pt2_hypspace* (4.1.2)	*no_overlap_pt1* (4.1.5)	*no_overlap_pt2* (4.1.6)	*spt1_in_pool* (4.1.3)	*spt2_in_pool* (4.1.4)	*map_usr_allowed* (4.2.1)	*frange_no_usr* (4.2.2)	*free_not_mapped* (4.2.3)	*pool_in_frange* (4.2.4)	*alloc_is_alloc* (4.2.5)	*hpt_phys2virt* (4.1.7)	*spt_phys2virt* (4.1.8)
hpt_pt1_hypspace	×	×	×	×	×	×				×			×
hpt_pt2_hypspace	×	×	×	×	×	×				×			×
no_overlap_pt1	×	×	×	×	×	×				×			×
no_overlap_pt2	×	×	×	×	×	×				×			×
spt1_in_pool	×	×	×	×	×	×				×			×
spt2_in_pool	×	×	×	×	×	×				×			×
(*map_usr_allowed*)	×	×	×	×	×	×	×		×	×	×	×	×
(*frange_no_usr*)	×	×	×	×	×	×	×	×	×	×	×	×	×
free_not_mapped	×	×	×	×	×	×			×	×	×		×
pool_in_frange	×	×	×	×	×	×				×			×
alloc_is_alloc	×	×	×	×	×	×			×	×	×		×
(*hpt_phys2virt*)	×	×	×	×	×	×	×		×	×	×	×	×
spt_phys2virt	×	×	×	×	×	×				×			×

FIGURE 4.5: Invariants Dependencies: for each invariant in a row, we mark with a cross all the properties used in the proof of its preservation over the ***unmap_all*** operation.

4.3 Specifications of the Effects of some Transitions

In this section, we introduce lemmas which describe the effects of the guest transition and the memory management operations on the global state. For each operation, we present two lemmas, one specifying the modified parts, one specifying the parts that remain identical.

An important point to make is that even if in both cases, a part of the memory is modified, the observable effects are not of the same nature for guest transition and memory management operations. Indeed, the guest transition changes the byte values of the memory accessible to the guests, while the memory operations change the rights attributed to them. In the next chapter, we present the abstract (or observable) state, in which we do not represent anymore the memory where SPT are kept, but only the access rights they define. We will see in Chapter 5 how the lemmas that we prove in this section allow to formally link the abstract with the concrete model.

4.3.1 Map

We specify the effects of the *map* operation in Lemma 4.3.1. It states that the virtual addresses within the virtual page just mapped are mapped and describes how, whereas the other mappings are not modified. The proof of this lemma requires eight of the invariant properties: the PT invariants, the invariant for translating physical addresses of the *pool*

to virtual addresses with the HPT, and the *free_not_mapped* invariant which is needed to preserve all the other invariants over the map operation.

Lemma 4.3.1 (Map Effects). If we map a virtual address va_{pg} to a physical address pa_{pg} both aligned to the size of a page, in the SPT of guest i, with rights r, and if this new mapping is allowed by the static permissions, then all the addresses in the virtual page at va_{pg} are mapped to physical addresses in the page at pa_{pg} with rights r, and the mappings for all the other do not change:
$\forall st \in St, \forall st^{new} \in St, \forall \boldsymbol{va}_{pg} \in Addr_{pg}$ and $\forall \boldsymbol{pa}_{pg} \in Addr_{pg}$,

Let $base_{\text{SPT}} = st.\sigma_{\text{HYP}}.vcpus(i).base_{\text{SPT}}$
$mem = st.\sigma_{\text{HW}}.mem$
$base_{\text{SPT}}{}^{new} = st^{new}.\sigma_{\text{HYP}}.vcpus(i).base_{\text{SPT}}$
$mem^{new} = st.\sigma_{\text{HW}}.mem^{new}$

If $hpt\_pt1\_hypspace$,
$hpt\_pt2\_hypspace(mem)$,
$no\_overlap\_pt1(mem, vcpus)$,
$no\_overlap\_pt2(mem, vcpus)$,
$spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$hpt\_phys2virt(mem, vcpus)$,
$free\_not\_mapped(mem, vcpus)$,
$st' = \boldsymbol{map}(mem, vcpus, i, \boldsymbol{va}_{pg}, \boldsymbol{pa}_{pg}, r)$,

Then, $\forall va \in Addr$:

$$va \notin [\boldsymbol{va}_{pg}, \boldsymbol{va}_{pg} + size_{page}[\Rightarrow pt(mem, base_{\text{SPT}})(va) = pt(mem', base_{\text{SPT}}{}')(va)$$
$$va \in [\boldsymbol{va}_{pg}, \boldsymbol{va}_{pg} + size_{page}[\Rightarrow \text{let } off < size_{page} \text{ s.t. } va = \boldsymbol{va}_{pg} + off, \; pt(mem', base_{\text{SPT}}{}')(va) = (\boldsymbol{pa} + off, r)$$

Proof. We only show the proof when the map definition falls into the second case (*allocate a PT1 before mapping*). The proof of the last case (*Case: PT1 is already present, add an entry in it*) is simpler, and follows the same scheme. In this proof we use the properties presented in the previous section, without naming which specifically, but giving the intuition:

Let $idx_1^{pg}, idx_2^{pg} \in Uint$ such that $va_{pg} = idx_1^{pg} \oplus idx_2^{pg}$
$idx_1, idx_2, off \in Uint$ such that $va = idx_1 \oplus idx_2 \oplus off$
(Virtual SPT base pointer:)
$vbase_{\text{SPT}} = phys2virt(base_{\text{SPT}})$
(Virtual page directory entry at index idx_1^{pg}:)
$vpde^{pg} = compute(vbase_{\text{SPT}}, \boldsymbol{idx}_1^{pg})$
(Physical page directory entry at index idx_1:)
$ppde = compute(vcpus(i).base_{\text{SPT}}, \boldsymbol{idx}_1)$

We know that $get\_v\_descr_1(mem, base_{\text{HPT}}, vpde) \in \{Fault\}$ from the definition of map. As $hpt\_phys2virt$ holds and $ppde \in perm.pool(i)$, the lemmas on equivalence of physical and virtual getters gives that $get\_descr_1(mem, ppde) \in \{Fault\}$ too. Meaning that va was not mapped before the map operation, by definition of PT (Definition 3.2.10. i.e. $pt(mem, base_{\text{SPT}}) = fault$.

Case $\boldsymbol{va} \notin [\boldsymbol{va}_{pg}, \boldsymbol{va}_{pg} + size_{page}[$ By Property 3.2.1, $\neg(idx_1 = idx_1^{pg} \wedge idx_2 = idx_2^{pg})$ (1).
We reason on the decomposition of va.

Case $idx_1 \neq idx_1^{pg}$ As *spt1_in_pool* and *hpt_phys2virt* hold, we can use the PT modification properties to state that the first and second level descriptor holding the path for va have not been modified. Therefore, from Definition 3.2.10:

$$pt(mem^{new}, base_{\mathrm{SPT}}) = pt(mem, base_{\mathrm{SPT}})$$

Qed.

Case $idx_1 = idx_1^{pg}$ Because *no_overlap_pt2* holds, setting a second level descriptor has no impact on getting a first level descriptor (PT modification properties), thus:

$$get\_descr_1(mem^{new}, ppde) = get\_descr_1(mem', ppde)$$

The descriptor at idx_1 in mem^{new} corresponds to the address a of the second level of PT just allocated:

$$get\_descr_1(mem^{new}, ppde) = a$$

From *free_not_mapped* property, the second level PT at address a is empty, thus all of its entries are faults.

From (1), $idx_2 \neq idx_2^{pg}$. We have proved that our invariant properties hold over the setting of the second level descriptor. In particular *spt2_in_pool* and *hpt_phys2virt* hold for mem' and $vcpus'$, so again, we can use the PT modification properties to state that the second level descriptor holding the path for va has not been modified: Let $ppte$ be the virtual page table entry at idx_2 in the PT at address a, and $ppte = compute(a, idx_2)$, we have:

$$get\_descr_2(mem^{new}, ppte) = get\_descr_2(mem', ppte) = Fault$$

Therefore $pt(mem^{new}, base_{\mathrm{SPT}})(va) = Fault$, meaning that:

$$pt(mem, base_{\mathrm{SPT}}) = pt(mem^{new}, base_{\mathrm{SPT}})$$

Qed.

Case $\boldsymbol{va} \in [\boldsymbol{va}_{pg}, \boldsymbol{va}_{pg} + size_{page}[$ We prove this case by contradiction. By Property 3.2.1:

$$idx_1 = idx_1^{pg} \wedge idx_2 = idx_2^{pg} \text{ (2)}$$
$$\wedge\ off < size_{page} \text{ (3)}$$

Given (2) (3) and the PT modification properties, similar reasoning as previous cases show that va is in the page just mapped. In particular $pt(mem', base_{\mathrm{SPT}}{}')(va_{pg}) \notin Fault$.

Case $pt(mem', base_{\mathrm{SPT}}{}')(va) = (a, r_1) \wedge \neg(r_1 \neq r)$ va is in the page just mapped i.e. with the rights r, which contradicts $r_1 \neq r$. Qed.

Case $pt(mem', spt')(va) = (a, \_) \wedge a \notin [pa, size_{page}[$ The inequality (3) allows to prove that va^{pg} mapped to pa implies $va^{pg}+off$ mapped to $pa+off$. Yet $pa+off \in [pa, size_{page}[$, which leads to a contradiction. Qed.

□

The previous lemma specifies how the SPT of the concerned guest is affected by the map operation, next lemma specifies which region of memory is *not affected* by the map operation.

Lemma 4.3.2 (Map Unchanged)**.** If a physical address pa is not in a private or shared region, nor in a $pool$ of a guest i, then the byte value at pa is left unchanged by the map operation on that guest SPTs:
$\forall st \in St, \forall i, k \in Idx$ s.t. $k \neq i, \forall va_{pg} \in Addr_{pg}, \forall pa \in Addr,$

If $hpt\_pt1\_hypspace$,
$hpt\_pt2\_hypspace(mem)$,
$no\_overlap\_pt1(mem, vcpus)$,
$no\_overlap\_pt2(mem, vcpus)$,
$spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$hpt\_phys2virt(mem, vcpus)$,
$free\_not\_mapped(mem, vcpus)$,
$st' = \boldsymbol{map}(mem, vcpus, i, \boldsymbol{va}_{pg}, \boldsymbol{pa}_{pg}, r)$,
$(pa \in perm.priv(\_) \vee pa \in perm.shared(\_,\_) pa \in perm.pool(k))$,

Then $st'.\sigma_{\mathrm{HW}}.mem(pa) = st.\sigma_{\mathrm{HW}}.mem(pa)$

4.3.2 Unmap

Similarly as for the map operation, Lemmas 4.3.3 and 4.3.4 specify which parts of memory are modified and how under the $unmap$ operation.

Lemma 4.3.3 (Unmap Effects)**.** If we unmap an aligned virtual address va_1 from the current SPT of a guest i, then all the mappings at addresses not in the page of base va_{pg} are unchanged. Whereas all the mappings in the page of base va_{pg} are removed.
$\forall st \in St, \forall st^{new} \in St, \forall \boldsymbol{va}_{pg} \in Addr_{pg}$ and $\forall \boldsymbol{pa}_{pg} \in Addr_{pg},$

Let $base_{\mathrm{SPT}} = st.\sigma_{\mathrm{HYP}}.vcpus(i).base_{\mathrm{SPT}}$
$mem = st.\sigma_{\mathrm{HW}}.mem$

If $hpt\_pt1\_hypspace$,
$hpt\_pt2\_hypspace(mem)$,
$no\_overlap\_pt1(mem, vcpus)$,
$no\_overlap\_pt2(mem, vcpus)$,
$spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$spt\_phys2virt(mem, vcpus)$,
$pool\_in\_frange(mem, vcpus)$,
$\boldsymbol{va}_{pg} \notin vcpus(i).frange$,
$st^{new} = \boldsymbol{unmap}(mem, vcpus, i, \boldsymbol{va}_{pg})$,

Then, $\forall va \in Addr$:

$$\begin{array}{lcl} va \notin [va_1, va_1 + pg\_size[& \Rightarrow & pt(mem, spt)(va) = pt(mem', spt')(va) \\ va \in [va_1, va_1 + pg\_size[& \Rightarrow & pt(mem', spt')(va) = Fault \end{array}$$

Lemma 4.3.4 (Unmap Unchanged)**.** If a physical address pa is not in any private or shared region, then the byte value at pa is left unchanged by $unmap$:

$\forall st \in St, \forall i, j, k \in Idx, \forall va_{pg} \in Addr_{pg}, \forall pa \in Addr,$
Let $base_{\text{SPT}} = st.\sigma_{\text{HYP}}.vcpus(i).base_{\text{SPT}}$
$mem = st.\sigma_{\text{HW}}.mem$

If $hpt\_pt1\_hypspace$,
$hpt\_pt2\_hypspace(mem)$,
$no\_overlap\_pt1(mem, vcpus)$,
$no\_overlap\_pt2(mem, vcpus)$,
$spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$spt\_phys2virt(mem, vcpus)$,
$pool\_in\_frange(mem, vcpus)$,
$free\_not\_mapped(mem, vcpus)$,
$st^{new} = \boldsymbol{unmap}(mem, vcpus, i, \boldsymbol{va}_{pg})$,
$(pa \in perm.priv(\_) \vee pa \in perm.shared(\_, \_) \vee pa \in perm.pool(k))$,

Then $st^{new}.\sigma_{\text{HW}}.mem(pa) = mem(pa)$

4.3.3 Unmap All

After an $unmap\_all$ operation on the SPT of a guest, all the mappings in the forbidden range are unchanged, whereas all the mappings outside of the forbidden range are removed. $frange\_no\_map$ ensures that the addresses of the forbidden range are not mapped with user rights, hence we know that after the map operation, no address is mapped with user rights in the SPT of the concerned guest (Lemma 4.3.5).

Lemma 4.3.5 (Unmap All)**.** The $unmap\_all$ operation removes all the user mappings in the SPT of a guest.
$\forall st \in St, \forall st^{new} \in St, \forall \boldsymbol{va}_{pg} \in Addr_{pg}$ and $\forall \boldsymbol{pa}_{pg} \in Addr_{pg}$,
Let $base_{\text{SPT}} = st.\sigma_{\text{HYP}}.vcpus(i).base_{\text{SPT}}$
$mem = st.\sigma_{\text{HW}}.mem$

If $hpt\_pt1\_hypspace$,
$hpt\_pt2\_hypspace(mem)$,
$no\_overlap\_pt1(mem, vcpus)$,
$no\_overlap\_pt2(mem, vcpus)$,
$spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$spt\_phys2virt(mem, vcpus)$,
$pool\_in\_frange(mem, vcpus)$,
$free\_not\_mapped(mem, vcpus)$,
$frange\_no\_map(mem, vcpus)$
$st^{new} = \boldsymbol{unmap\_all}(mem, vcpus, i)$,

Then, $\forall va \in Addr, va \notin Map_{\text{USR}}(st^{new}.\sigma_{\text{HW}}.mem, base_{\text{SPT}})$

The lemma 4.3.6 is similar as the lemmas for map and $unmap$.

Lemma 4.3.6 (Unmap All Unchanged)**.** If a physical address pa is not in any private or shared region, then the byte value at pa is left unchanged by $unmap$:
$\forall st \in St, \forall i, j, k \in Idx, \forall va_{pg} \in Addr_{pg}, \forall pa \in Addr,$
Let $base_{\text{SPT}} = st.\sigma_{\text{HYP}}.vcpus(i).base_{\text{SPT}}$
$mem = st.\sigma_{\text{HW}}.mem$

If $hpt\_pt1\_hypspace$,
$hpt\_pt2\_hypspace(mem)$,
$no\_overlap\_pt1(mem, vcpus)$,
$no\_overlap\_pt2(mem, vcpus)$,
$spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$spt\_phys2virt(mem, vcpus)$,
$pool\_in\_frange(mem, vcpus)$,
$st^{new} = \boldsymbol{unmap\_all}(st, i)$,
$k \neq i$,
$(pa \in perm.priv(\_) \vee pa \in perm.shared(\_, \_) \vee pa \in perm.pool(k))$
Then $st^{new}.\sigma_{\mathrm{HW}}.mem(pa) = mem(pa)$

4.3.4 Guest Transition

The specification of the guest transition cannot be as accurate as for a hypervisor transition, because it is performed by the guest, and must be expressed independently of the particular guest considered. Basically, we only specify the parameters which stay unchanged: all the SPT regions, the private regions of the guest which does not run, and the shared region which are not writable by the current guest.

Lemma 4.3.7 (Guest Trans Unchanged)**.** If a physical address pa is not in the private or send region of the current guest, then the byte value at pa is left unchanged by $guest\_trans$:
$\forall st \in St, \forall j, k \in Idx, \forall va_{pg} \in Addr_{pg}, \forall pa \in Addr$,
Let $curr = st.\sigma_{\mathrm{HYP}}.curr$
$base_{\mathrm{SPT}} = st.\sigma_{\mathrm{HYP}}.vcpus(i).base_{\mathrm{SPT}}$
$mem = st.\sigma_{\mathrm{HW}}.mem$
If $spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$map\_usr\_allowed(mem, vcpus)$,
$frange\_no\_usr(mem, vcpus)$,
$st \xrightarrow{GuestTrans(o)} st'$,
$k \neq curr$,
$(pa \in perm.priv(k) \vee pa \in perm.shared(k, j) \vee pa \in perm.spt(j))$,
Then $\boldsymbol{st'}.\sigma_{\mathrm{HW}}.mem(pa) = st.\sigma_{\mathrm{HW}}.mem(pa)$

Proof. We give the proof sketch of this lemma. Invariants $spt1\_in\_pool$ and $spt2\_in\_pool$ ensure that the SPT of the guest are in its pool. As $map\_usr\_allowed$ holds, we know that the $pool$ regions are not mapped with user rights. In particular, the SPTs do not map themselves.

Consequently, we can apply the Axiom 3.5.1, which states that the memory is modified only on addresses mapped with user rights in the current guest SPT. From the property $map\_usr\_allowed$, we know that these addresses correspond to the addresses in the private or shared region of the current guest, hence the result. □

The fact that the SPT regions are not modify allow to establish that the mappings stay unchanged. This is not true for the memory operations.

Lemma 4.3.8 (Guest Trans Same Mappings). $guest\_trans$ does not modify the mappings: $\forall st \in St, \forall j, k \in Idx, \forall va_{pg} \in Addr_{pg}, \forall pa \in Addr$,

Let $curr = st.\sigma_{\mathrm{HYP}}.curr$
$base_{\mathrm{SPT}} = st.\sigma_{\mathrm{HYP}}.vcpus(i).base_{\mathrm{SPT}}$
$mem = st.\sigma_{\mathrm{HW}}.mem$
If $spt1\_in\_pool(mem, vcpus)$,
$spt2\_in\_pool(mem, vcpus)$,
$map\_usr\_allowed(mem, vcpus)$,
$frange\_no\_usr(mem, vcpus)$,
$st \xrightarrow{GuestTrans(o)} st'$,
Then $pt(mem', base_{\mathrm{SPT}})(va) = pt(mem, base_{\mathrm{SPT}})(va)$

4.4 Conclusion

We have presented all the invariants that we need to prove the refinement over the guest transition and the MMU transitions.

In the next chapters, we use the following definition instead of stating explicitly which invariant we need:

Definition 4.4.1 (Well-Formed State). A state is well-formed if it respects all the invariants: $\forall st \in St, wf(st) \Leftrightarrow$

$hpt\_pt1\_hypspace \wedge$	*(The first level HPT is included in the hypervisor space)*
$hpt\_pt2\_hypspace(st.\sigma_{\mathrm{HW}}.mem) \wedge$	*(The second level HPTs are included in the hypervisor space)*
$no\_overlap\_pt1(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(The first level SPT of a guest does not overlap with its second level SPTs)*
$no\_overlap\_pt2(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(The second level SPTs of a guest does not overlap with its second level SPTs)*
$spt1\_in\_pool(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(The first level SPT of a guest is in its pool)*
$spt2\_in\_pool(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(The second level SPTs of a guest are in its pool)*
$hpt\_phys2virt(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(Specification of how the pool is mapped in the HPT)*
$spt\_phys2virt(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(Specification of how the pool is mapped in the SPTs)*
$map\_usr\_allowed(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(Physical addresses mapped in the SPT of a guest are allowed for that guest)*
$pool\_in\_frange(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(The pool is mapped by virtual addresses in the forbidden range)*
$free\_not\_mapped(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(A page listed as free is not mapped)*
$alloc\_is\_alloc(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus) \wedge$	*(A page not listed as free is mapped)*
$frange\_no\_usr(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus)$	*(The forbidden range is not mapped with user rights)*

We have proved the preservation of these invariants over the guest transition, the map, $unmap$ and $unmap\_all$ operations. In particular, it means that a well-formed state stays well-formed after these operations.

This definition does not include the $spt\_curr\_pt(st)$, which is broken before the map operation and reestablished just after. For all $st \in St$, we note $wf^{+}(st)$ for $wf(st) \wedge spt\_curr\_pt(st)$.

We have proved the effects of the guest transition, the map, $unmap$ and $unmap\_all$ operations on the global state. We will use these lemmas to reason on the effects of these operation on the observable state in Chapter 5, Section 5.5.

4.5 Key Points

- We have presented three groups of invariants:
  - the invariants about PT well-formedness,
  - the invariants that allow to translate physical to virtual addresses,
  - the specific invariants, which allow to prove the preservation of the two first groups of invariants over the transitions of the system.
- We have formally proved the preservation of all the invariants over all the memory operations, and over the guest transition.
- We have presented the dependencies between these invariants in the preservation proofs.
- We have formally proved, for each memory operation and for the guest transition, the specification of its effects on the state components.

Chapter 5

Abstract Model of the Hypervisor

Contents

In Chapter 3, we have presented the concrete model, which is close to the implementation. In this chapter, we present a corresponding abstract model, which is close to the specifications.

In the concrete state, all of the guests data are stored in the same memory. The permissions (Definition 3.3.1) define, for each guest, which region of memory is accessible and how:

- A region can be *private*, meaning that only one guest can access it.

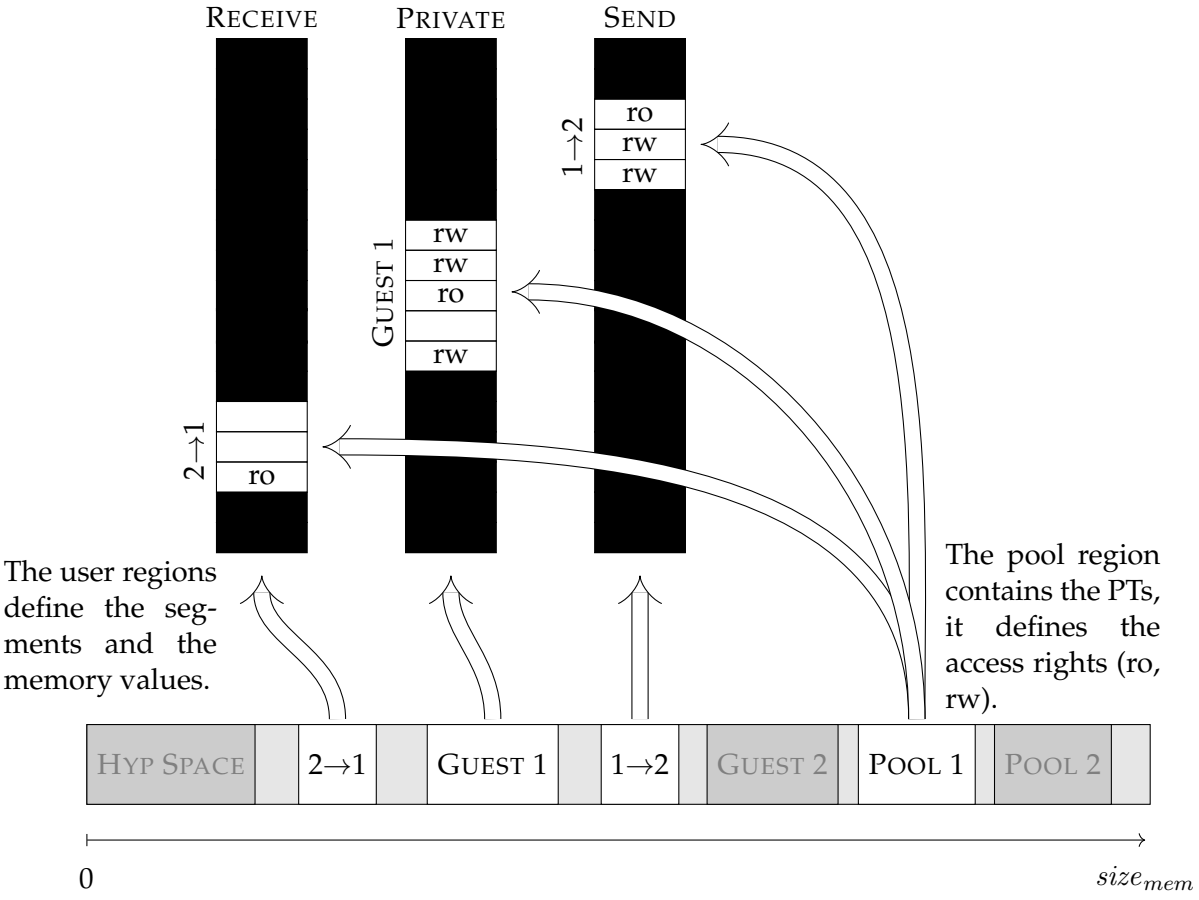


FIGURE 5.1: Abstraction of the Memory for Guest 1

- A region can be *shared* between two guests i and j, in this case the region is either a write buffer from i to j or the contrary.

- A region can be the *pool* of a guest i, meaning that it holds the SPT of guest i, and cannot be accessed by any guest.

However the access rights are managed through SPTs. It is not obvious that the rights defined by the SPTs respect the permissions, all the more that the SPTs are themselves stored in memory. Isolation between guest's regions of memory in the concrete state is therefore not obvious.

In the abstract state transition system, parts of memory which are isolated from each other are represented by distinct structures. A transition only impacts some structures, therefore isolation becomes apparent.

We illustrate the abstraction of memory in Figure 5.1. On the bottom of the figure, we have represented the physical memory, with the regions defined by the permissions. The regions in white represent what is *observable* for guest 1: its receive buffer with guest 2, its private region, its send buffer to guest 2 and its pool. Each user region (receive buffer, private region and send buffer) is represented in a different segment in the abstract state. As we will develop in Section 5.1 we use the concrete permissions to define which addresses of the abstract segments are defined, and we use the memory byte values at these addresses to fill the abstract memory cells. The concrete pool region holds the SPTs,

it defines, for each address of the segment, which rights are associated to it (rw or ro). The rights are stored in each abstract memory cell, as we will see in Definition 5.1.2.

In Sections 5.1, 5.2 and 5.3, we present the abstract state, the abstraction function and the abstract transitions. In Section 5.4 we present the security properties and their proof.

Last but not least, we must show that the concrete state transition system is a refinement of the abstract state transition system. To do so we must show that, if an abstract state $st^{\#}$ is the abstraction of a concrete state st, then the abstract state resulting from a transition from $st^{\#}$ is the abstraction of the concrete state resulting from a transition from state st, as illustrated in Figure 1.4. For example, in the abstract state transition system, the guest transition made by guest 1 does not modify its receive buffer, following what is specified by the permissions. However, if the concrete state is not well-formed, some addresses of the receive buffer of guest 1 might be mapped in RW by its SPT, meaning that the guest could modify its receive buffer during the concrete guest transition. The abstract transition would therefore not correspond to the concrete transition.

In Chapter 4, we have presented invariant properties of the concrete system. We have specified the effects of transitions on states verifying the invariant properties. We use the results of Chapter 4 in order to show that this kind of misbehaviour cannot happen, to achieve the proofs of refinement in Section 5.5.

5.1 Abstract State

This section presents the abstract model used to prove that SPTs provide memory isolation between guests. The abstract state transition system describes guests which have a private region of memory, a read and a write buffer shared with every other guest and a set of registers owned only by itself. The abstract state transition system corresponds to what is *observable* for each guest. We give the definition of the abstract state at the end of this section (Definition 5.1.5).

5.1.1 Memory Cells

First of all we describe the representation of the memory in the abstract state. In the concrete state, there is one memory shared by all the guests. Each memory cell holds a byte. In the abstract state, each guest has its own representation of the memory, as illustrated in Figure 5.1. More precisely, each guest has several segments of memory, one private, and several shared segments. Each cell of a segment is the association of a *byte* value and a set of *tags*, where a tag is a pair of a virtual address and a right (Definition 5.1.1). The set of tags of a physical address gathers all the virtual addresses that map this physical address in the SPT of a guest, with the rights associated to it.

Definition 5.1.1 (Tag)**.**

$$Tag = \begin{cases} va : Addr \\ rights : Rights \end{cases}$$

Definition 5.1.2 (Cell)**.**

$$Cell = \begin{cases} byte : Byte \\ tags : \{Tag\} \end{cases}$$

In particular, it means that, if a concrete physical address a is in the send buffer of guest i, shared with guest j, and if this address is mapped with RW rights in the SPT of guest i and with RO rights in the SPT of guest j, it will be represented twice in the abstract model:

- In a send segment of guest i, by a cell whose tags contain the element $< \_, rw >$.
- In a receive segment of guest j, by a cell whose tags contain the element $< \_, ro >$.

We will formalize this with the abstraction functions in Section 5.2.2.

Note that, in order to express our properties of confidentiality and integrity, we do not need to reason at the granularity of addresses, because we want to prove that a guest can only write in its private and send segments, and read from its receive segments. Therefore, as far as properties are concerned, a *cell* could just be a byte, and the access rights could just be set for a whole segment.

If we keep some tags for each cell, it is because we need our abstract system to be *deterministic*, as we have explained in Section 2.4. In particular, recall from the Section 3.5.1 that on the concrete model, the guest may only modify the memory cells for which it has RW access. If we represent a cell as a mere byte, then two concrete states with the same byte value in memory and different access rights would be projected to the *same* abstract state. They could transit to two states with different byte values, i.e. which would be projected to two *different* abstract states. If we keep such an imprecise representation of memory, there exist abstract states from which we cannot decide which transition to take, i.e. the abstract state transition system is not deterministic.

Therefore, we keep in the $tags$ field of the *cell* structure enough information to distinguish two concrete systems which do not map a physical address the same way.

5.1.2 Guest State

In the abstract state, its guest has its own state, which represents its own representation, or *view*, of the concrete state. The abstract guest state has four components:

- Some abstract registers.
- A private segment, in which the guest owner can write.
- As many send segments as the number of guests, in which the guest can write data to be shared with another guest.
- As many receive segments as the number of guests, in which the guest can read data written by another guest.

We have presented the permissions (Definition 3.3.1) in the concrete model, the segments of the abstract model result from the permissions, as we will see in Section 5.2.2. A segment is a function which associate to each address a cell or nothing:

Definition 5.1.3 (Segment)**.**

$$Seg : Addr \to (Cell + None)$$

We present the abstract registers directly with the abstraction function (Definition 5.2.1). We give the definition of the guest abstract state below:

Definition 5.1.4 (Guest Abstract State)**.**

$$St_G = \begin{cases} aregs : Regs_{abs} & \textit{(Registers)} \\ priv : Seg & \textit{(Private segment)} \\ send : Idx \to Seg & \textit{(Send segments)} \\ rcv : Idx \to Seg & \textit{(Receive segments)} \end{cases}$$

5.1.3 Whole State

Formally, the abstract state is composed of the index of the guest currently running, and all the states of all the guests:

Definition 5.1.5 (Abstract State)**.**

$$St^{\#} = \begin{cases} curr : Idx & \textit{(Index of the current guest)} \\ guests : Idx \to St_G & \textit{(State of all the Guests)} \end{cases}$$

5.2 Abstraction

In this section, we define the *abstraction function*, or *view function*, which takes a concrete state and returns an abstract state. To do so we define the projection of the abstraction function for each field of the state. The current guest field in the abstract state is equal to the current guest field in the concrete state. The view of an abstract guest state is itself split into the views of each of its abstract field.

We define the abstraction function for each field below, and finally, we give the definitions of $view_G$ (Definition 5.2.7) and $view$ (Definition 5.2.8) at the end of this section.

5.2.1 Registers

The registers that a guest may observe are the concrete virtual registers stored in the concrete guest state ($regs_{\text{virt}}$, Definition 3.4.8), except for the registers related to the generic interrupt controller ($regs_{\text{gic}}$) which have been entirely abstracted:

Definition 5.2.1 (Abstract Registers)**.**

$$Regs_{abs} = \begin{cases} mode : Mode \\ apsr : Apsr \\ masks : Masks \\ regs_{\text{core}} : vRegs_{\text{core}} \\ regs_{\text{bnk}} : aRegs_{\text{bnk}} \\ regs_{\text{mmu}} : vRegs_{\text{mmu}} \end{cases}$$

We have represented the virtual CPSR in three parts:

- The bits corresponding to the mode.
- The bits corresponding to the APSR.
- The bits corresponding to the masks.

Similarly, the abstract banked registers $aRegs_{\text{bnk}}$ have the same type as the concrete $vRegs_{\text{bnk}}$, except that the SPSRs are split in three parts as well. We note $view_{bnk}$ the abstraction from $vRegs_{\text{bnk}}$ to $aRegs_{\text{bnk}}$.

The abstraction function for registers has the following type:

$$view_{regs} : (Idx \to vCPU) \times Idx \to Regs_{abs}$$

and is defined in Definition 5.2.2. Similarly to the function $mode$, the functions $apsr$ and $masks$ extract the bits of the CPSR corresponding respectively to the APSR and the mask bits. As we have explained in Section 4.2.5, the function $mode$ never raises $none$.

Definition 5.2.2 (Register Abstraction). $\forall vcpus \in (Idx \to vCPU)$ and $i \in Idx$,

$$view_{regs}(vcpus, i) = \begin{cases} mode(vcpus(i).regs_{\text{virt}}.cpsr) \\ apsr(vcpus(i).regs_{\text{virt}}.cpsr) \\ masks(vcpus(i).regs_{\text{virt}}.cpsr) \\ vcpus(i).regs_{\text{virt}}.vregs_{\text{core}} \\ view_{bnk}(vcpus(i).regs_{\text{virt}}.vregs_{\text{bnk}}) \\ vcpus(i).regs_{\text{virt}}.vregs_{\text{mmu}} \end{cases}$$

5.2.2 Segments

The segments are functions from addresses to cells, they represent the memory of the abstract state. If a guest has permissions to a physical address, then one of its segment is defined on this physical address. The cell associated to this address in the segment is composed of the byte present in concrete memory at this address, and a set of tags, which keep track of which virtual address maps this physical address in the SPT of that guest, and with which rights.

More precisely, the tags of a guest i in $vcpus$ associated to a physical address pa are the couples of virtual addresses va and rights r such that the address va is mapped to the address pa with rights r in the current SPT of guest i. Notice that we consider a *set* of tags and not a unique tag because several virtual addresses might map the same physical address in the concrete model.

The $tags$ function has the following type:

$$tags : Mem \times (Idx \to vCPU) \times Idx \times Addr \to \{Tag\} + Oob$$

We define it below:

Definition 5.2.3 (Tags Function). $\forall mem \in Mem$, $\forall vcpus \in (Idx \to vCPU)$, $\forall i \in Idx$ and $\forall \boldsymbol{pa} \in Addr$,
$tags(mem, vcpus, i, pa) =$

Let $tags' = \{\}$
If $pa \in [0, size_{mem}[$
  For $va = 0...2^{32}$
    If $pt(mem, base_{\text{PT}})(\boldsymbol{va}) = (\boldsymbol{pa}, r)$, $r \in \{rw, ro\}$,
    Then $tags' = tags' \cup (\boldsymbol{va}, r)$
  End For
  return $tags'$
Else **return** oob

It does not return Oob for addresses in the memory range.

Having a function is important because it proves the existence of a set of tags in relation with the concrete state components. However, for commutation proofs, we are only interested in the effects of such a function, which we describe in Lemma 5.2.1.

Lemma 5.2.1 (Tags). $\forall mem \in Mem, \forall vcpus \in (Idx \to vCPU), \forall pa \in [0, size_{mem}[$. Let $table$ be the current SPT of the guest i: $table = pt(mem, vcpus(i).base_{\text{SPT}})$. The tags of the guest i in $vcpus$ for an address pa, can be defined as follows:

$$tags(mem, vcpus, i, pa) = \{\langle va, r\rangle | table(va) = \langle pa, r\rangle\}$$

We present below the abstraction functions for the private, send and receive segments. We do not present the whole implementation of the function, rather, we give the definition for a particular address of the resulting segment.

Private Segment

The function of abstraction of the private segment, $view_{priv}$ takes as argument the memory, the permissions, the vCPUs and an index of vCPU and returns an abstract segment:

$$view_{priv} : Mem \times Perm \times (Idx \to vCPU) \times Idx \to Seg$$

If an address is not in the private region of the guest (as defined by the permissions, Definition 3.3.1), then the private segment is a $None$ at this address. If an address is in the private region of the guest, then there is a cell at this address in the abstract private segment of the guest. The byte of the cell is the byte value of the concrete memory at this address whereas the set of tags is given by the SPTs located in the pool of that guest.

Definition 5.2.4 (Private Segment Abstraction)**.** $\forall mem \in Mem$, $\forall vcpus \in (Idx \to vCPU)$, $\forall i \in Idx$ and $\forall \boldsymbol{pa} \in Addr$,
$\boldsymbol{view_{priv}}(mem, perm, vcpus, i)(\boldsymbol{pa}) =$

If $pa \in perm.priv(i)$,
Then Let $b = mem(\boldsymbol{pa})$,
$tags = tags(mem, vcpus, i, \boldsymbol{pa})$,
return $\langle \boldsymbol{b}, \boldsymbol{tags} \rangle$
Else **return** $\boldsymbol{none}$

Note that the $tags$ function is invoked on addresses of the private region of a guest, meaning on addresses located in memory. Therefore the function $tags$ does not raise Oob when called from the $view_{priv}$ function. The same applies to the views of shared segments defined in the next section.

Shared Segments

The function of abstraction of the shared segments, $view_{send}$ and $view_{rcv}$ both have the same type. They take as argument the memory, the permissions, the vCPUs and an index i of vCPU and return a function which associate, to each vCPU j, the shared segment with vCPU i.

$$view_{send} : Mem \times Perm \times (Idx \to vCPU) \times Idx \to (Idx \to Seg)$$

For each guest j, the send segment of guest i, shared with j, is defined just as the private segment is. It means that if an address pa is in the shared buffer from i to j ($pa \in perm.shared(i, j)$), then there is a cell at this address in the abstract send segment j of the guest i. The byte of the cell is the byte value of the concrete memory at this address whereas the set of tags is given by the SPTs located in the pool of guest i.

Definition 5.2.5 (Send Segments Abstraction)**.** $\forall mem \in Mem$, $\forall vcpus \in (Idx \to vCPU)$, $\forall i \in Idx$, $i, \boldsymbol{j} \in Idx$, and $\forall \boldsymbol{pa} \in Addr$.

$\boldsymbol{view_{send}}(mem, perm, vcpus, i)(\boldsymbol{j})(\boldsymbol{pa}) =$

If $pa \in perm.shared(i, \boldsymbol{j})$,
Then Let $b = mem(\boldsymbol{pa})$,
$tags = tags(mem, vcpus, i, \boldsymbol{pa})$,
return $\langle \boldsymbol{b}, \boldsymbol{tags} \rangle$
Else **return** $\boldsymbol{none}$

The receive segments abstraction is similar to the send segments abstraction, except that we abstract the physical addresses in the domain of $shared(j, i)$ instead of $shared(i, j)$.

Definition 5.2.6 (Receive Segments Abstraction)**.** $\forall mem \in Mem$, $\forall vcpus \in (Idx \to vCPU)$, $\forall i \in Idx$, $i, \boldsymbol{j} \in Idx$, and $\forall \boldsymbol{pa} \in Addr$.
$\boldsymbol{view_{rcv}}(mem, perm, vcpus, i)(\boldsymbol{j})(\boldsymbol{pa}) =$

If $pa \in perm.shared(\boldsymbol{j}, i)$,
Then Let $b = mem(\boldsymbol{pa})$,
$tags = tags(mem, vcpus, i, \boldsymbol{pa})$,
return $\langle \boldsymbol{b}, \boldsymbol{tags} \rangle$
Else **return** $\boldsymbol{none}$

The shared segments are therefore partially duplicated. A shared region of the concrete state ($perm.shared(i, j)$) is represented twice in the abstract model. First in the state of guest i, represented following what the guest i can observe, meaning as the j^{th} send segment of guest i. Secondly in the state of guest j, as its i^{th} receive segment.

In the two representations of this part of memory, the byte values are the same, but the tags can be different, as each guest will observe the rights defined by its own SPT.

5.2.3 Abstraction Function

We can now define the view of the guest $view_G : St \to (Idx \to St_G)$, and the view of the whole state $view : St \to St^{\#}$.

Definition 5.2.7 (View Guest)**.** $\forall st \in St$, $\forall i \in Idx$:
Let $mem = st.\sigma_{\mathrm{HW}}.mem$,
$vcpus = st.\sigma_{\mathrm{HYP}}.vcpus$,

$$view_G(st)(i) = \begin{cases} view_{regs}(vcpus, i) \\ view_{priv}(mem, perm, vcpus, i) \\ view_{send}(mem, perm, vcpus, i) \\ view_{rcv}(mem, perm, vcpus, i) \end{cases}$$

Definition 5.2.8 (View State)**.** $\forall st \in St$, $view(st)(i) = \begin{cases} st.curr \\ view_G(st) \end{cases}$

5.3 Abstract Transitions

We present here the abstractions of the transitions presented in Section 3.5. Figure 5.2 illustrates the correspondence between concrete and abstract transitions. In the left graph, we have split the restore transition into the two parts described in Section 3.5.4: *InjectIRQ* and *LoadState*. The abstract $Guest^{\#}$ transition is the view of the composition of the

LoadState, *Guest* and *SaveState* transitions. The $Hypervisor^{\#}$ transitions merely correspond to the concrete *Hypervisor* transitions. The abstract *Restore* transition corresponds to the first part of the concrete $Restore^{\#}$ transition: the $InjectIRQ$.

FIGURE 5.2: Correspondence between Transitions of the Concrete Level (left) and the Abstract Level (right)

Before presenting the transitions, we introduce the oracle, which is an extra argument of some of our transitions.

5.3.1 Oracle

Our abstract transition system is not deterministic. We first explain why we need it to be deterministic, then we explain how we make it deterministic.

Determinism for Confidentiality We have explained in details our methodology of proof by abstraction in the state of the art, in Section 2.4. In particular, we have given a formal definition of a general confidentiality property, and we have shown that when considering deterministic systems, a property could be transferred down to the concrete level (Lemma 2.4.2).

We explain it again less formally. In Figure 5.3, we have two start states, one on the top left and one on the bottom left. These states contain two guest states: the state of the current guest and the state of a guest j. The states are equals except on a "secret" part of guest j. We call such states *similar* states regarding the current guest. The confidentiality property states that the transitions possible from two similar states do not depend on the secret of j, i.e.that the transition modifies equally the two states. In Figure 5.3, we have represented with the same pattern the part of the state which are equal.

In Figure 5.4, we also depict a transition from two *similar* states, but this time in a non-deterministic system. As can be seen, confidentiality cannot be proved, because, in the general case, the transitions $trans'$ and $trans$ do not modify the states in the same way. The confidentiality property cannot be proved as it is, but the weaker confidentiality property which state that, if two states are *similar* and if a particular transition t is triggered, then the resulting states are similar, can be prove. Therefore, we add an extra parameter, an oracle, that enable to decide which transition to take, and thus makes the system deterministic. Then we state our property starting from two similar states and two equal oracles.

Note that determinising the system makes the property weaker, because we *assume* that the difference in guest j's secrets do not affect the transition taken. We will develop this point latter, when we introduce the secrets and the oracles for our particular case.

We also underline that determinising the system is a sufficient condition to prove our property and transferring it to the concrete model, but it is not a necessary condition.

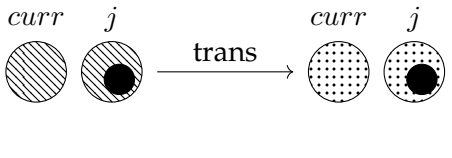


FIGURE 5.3: Confidentiality - Deterministic System

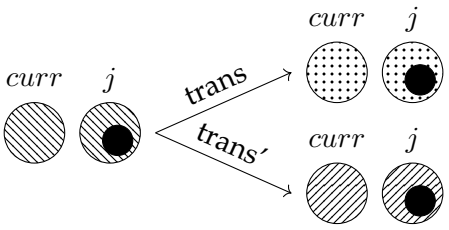
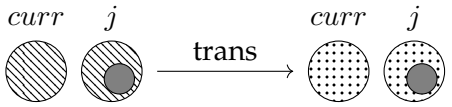
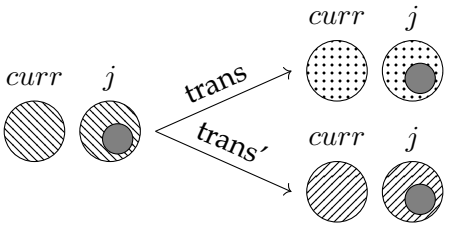


FIGURE 5.4: Confidentiality - Non Deterministic System





For example, suppose that the non-determinism only affects the secrets of guest j, then non-determinism is not an issue.

Presentation of the Three Oracles As explained in Section 3.5.1, in the concrete system, we have added an external oracle as an extra parameter for the guest transition, to decide whether an IRQ is raised during the transition, and therefore to make the system deterministic. This is an oracle that we keep in our abstract model.

In the abstract system, we have not specified every parts of the model. For some transition, we have not included in our model sufficient information to decide its behavior. Therefore, in order to keep a deterministic system, we add two more oracles, which make it possible to:

1. To decide which guest is to be run on a schedule.
2. To decide whether an IRQ is to be injected before restoring the guest.

It means that we reason with three extra arguments which make the system deterministic:

1. A guest to be run next ($nxt \in Idx$), used in the schedule transition.
2. The optional registers corresponding to the IRQ injection ($oirq \in Aregs$), used in the restore transition.
3. The optional raised IRQ ($oirq \in Oracle$), as defined in the concrete state, Section 3.5.1, used in the guest transition.

Correspondence with the Concrete State As we will develop in Section 5.5, our goal is to establish a formal correspondence between the concrete and abstract transitions. As depicted in Figure 5.5, we want to show that, if there exist a transition from concrete state

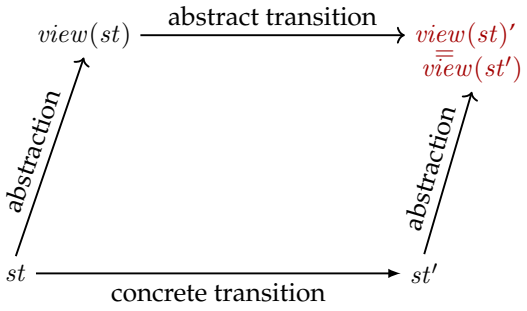


FIGURE 5.5: Commutation Diagram

st to st', and if there is a view of st. Then there is a view of st' and this view is equal to the result of the abstract transition from $view(st)$.

Therefore, not only must we make the abstract system deterministic, but we must also be able to link it to the concrete model, i.e. if two transitions are possible, we must choose the one that corresponds to the concrete transition, i.e. that "close" the commutation diagram. We develop below how we chose the oracles.

Concerning the third oracle, $oirq$, we merely take the same as for the concrete model, as we will develop when presenting the guest transition in Section 5.3.2.

The first and second oracles correspond to the results of some computation that can be made on the concrete level but not on the abstract level, because the information needed for the computation has been abstracted. We therefore use the result of the computation on the concrete level as an oracle. Figure 5.6 illustrates the commutation of the schedule transition with the oracle. The concrete schedule transition has an algorithm which chooses which guest is to be run next, and modifies the current guest (i.e. the field $st.\sigma_{\text{HYP}}.curr$) accordingly. We give the new current guest as an argument of the abstract scheduling transition. We do the same for the restore transition and the $ovirq$ oracle.

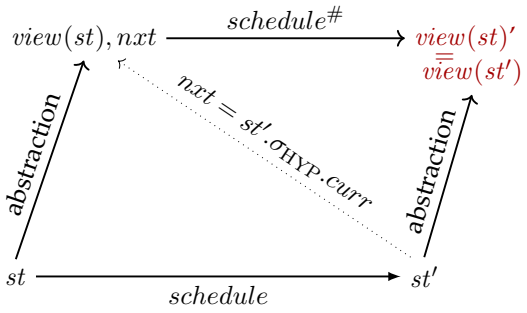


FIGURE 5.6: Oracle for Scheduling

In our proof of confidentiality, we compare the execution from two *similar* states, with the same oracle. We therefore make the hypothesis that in the concrete state, two similar states would schedule the same guest (resp. inject the same IRQ). We explain the extend of these hypothesis in Section 5.4.2, when we introduce the concept of similarity (Definitions 5.4.1 and 5.4.2).

5.3.2 Guest Transition

We have designed the abstract guest transition in such a way that some properties hold intrinsically, e.g. a guest does not modify its receive segments, on which it only has read access.

We divide the run into two steps:

- First, the current guest executes, modifying its registers, its private and send regions.
- Secondly, the modifications are reflected on the other guests. Basically, the values of the current guest send regions are copied to the receive regions of other guests. Such that, at the end of the synchronization, the guest send segments have the same values as the corresponding receive segments of other guests.

We present separately the two part of the function, before giving the definition of the guest transition at the end of this section.

Guest Run

A concrete state is well-formed if and only if it verifies all the system invariants defined in Chapter 4 (Definition 4.4.1). We introduce notations related to well-formed state in the following definition.

Definition 5.3.1 (Well-formed Abstract Guest States)**.** We denote by $\widehat{St}_G^+$ the set of guest state which are the image of a well-formed global state under $view_G$:

$$\widehat{St}_G^+ = \{\sigma_G \in St_G \mid \exists st \in St, wf^+(st) \wedge view_G(st) = \sigma_G\}$$

Definition 5.3.2 (Well-formed Abstract States)**.** Similarly, we denote by $\widehat{St}^{\#+}$ the set of abstract states which are the image of a well-formed global state under $view$:

$$\widehat{St}^{\#+} = \{st^{\#} \in St^{\#} \mid \exists st \in St, wf^+(st) \wedge view(st) = st^{\#}\}$$

We consider the relation run and we show that this relation is total and functional on the subset $\widehat{St}_G^+$ of St_G, i.e. that it is a function on $\widehat{St}_G^+$. The relation run[1] has the following type:

$$\underset{run}{\approx} : (St_G \times Oracle) \times (Regs_{abs} \times Seg \times (Idx \to Seg))$$

Definition 5.3.3 (Run Relation)**.** An abstract guest state is in relation with a tuple of registers, private and send segments if there exists a well-formed concrete state such that:

$\forall \sigma_G \in St_G, o \in Oracle, regs_{abs} \in regs_{abs}, priv \in Seg, send \in (Idx \to Seg)$,
$(\sigma_G, o) \underset{run}{\approx} (regs_{abs}, priv, send) \Leftrightarrow$

[1] In the implementation, the Definition 5.3.3 is more precise, because we only require some registers to be equals. Therefore Definition 5.3.5 is also more precise in the implementation. We do not put all the details here for clarity's sake.

$$
\begin{array}{ll}
\exists st \in St, wf^{+}(st), & \\
view_G(st, \sigma_G.curr) = \sigma_G, & [1] \\
st \xrightarrow{LoadState} \xrightarrow{GuestTrans(o)} \xrightarrow{SaveState} st^{new}, & [2] \\
send = view_G(st^{new}, i).send, & \\
priv = view_G(st^{new}, i).priv, & [3] \\
regs = view_G(st^{new}, i).regs &
\end{array}
$$

Figure 5.7 illustrates Definition 5.3.3. Several concrete states may be abstracted to the same state ([1]). We justify that the relation is functional by showing that if several concrete states verify [1], they all project to the same abstract state by [2] and [3]. More precisely, we show that two concrete states have the same view of guest i if and only if they are *equivalent for guest* i (Definition 5.3.6). Then that this relation is preserved by [2] (Lemma 5.3.1), so that [3] projects all these equivalent concrete states to the same abstract state.

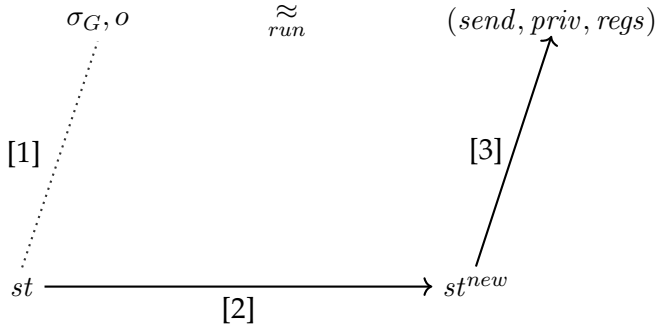


FIGURE 5.7: Run Relation

Definition 5.3.4 (Same User Region)**.** Two states have the same user regions if their memory is equal on user regions.

$$
\begin{array}{ll}
\forall st_1, st_2 \in St, same\_user\_region(st_1, st_2, i) \Leftrightarrow & \forall a \in user\_region(i), \\
& st_1.mem(a) = st_2.mem(a)
\end{array}
$$

Definition 5.3.5 (Same User Registers)**.** Two states have the same user registers iff their view of abstract registers is equal.

Let $st_1, st_2 \in St$,

$$
same\_user\_regs\_hyp(st_1, st_2) \Leftrightarrow
$$

$$
view_{regs}(st_1.\sigma_{\text{HYP}}.vcpus(i).regs_{\text{virt}}) = view_{regs}(st_2.\sigma_{\text{HYP}}.vcpus(i).regs_{\text{virt}})
$$

Definition 5.3.6 (Equivalence of Concrete States for a Guest)**.** . Two concrete states are equivalent regarding guest i if they have the same map for the SPT of guest i and the same virtual registers for i, and the same mapping defined.

$\forall st_1, st_2 \in \widehat{St}^{+}$,

$$
\begin{array}{ll}
st_1 \underset{i}{\approx} st_2 \Leftrightarrow & same\_map(st_1.mem, st_2.mem, st_1.sthyp.vcpus(i).base_{\text{SPT}}, st_2.sthw.base_{\text{PT}}), \\
& same\_user\_regs\_hyp(st_1, st_2), \\
& same\_user\_region(st_1, st_2)
\end{array}
$$

Lemma 5.3.1 (Same Abstract State). Two concrete and well-formed states are equivalent for a guest iff they have the same view of that guest.

$$\forall st, st' \in \widehat{St}^{+}, view(st, i) = view(st', i) \Leftrightarrow st \underset{i}{\approx} st'$$

Proof. The justification for $same\_user\_regs\_hyp$ comes directly from Definition 5.3.5.

The justification for $same\_map$ and $same\_user\_region$ comes from Definitions 5.2.4, 5.2.5 and 5.2.6, from Invariant 4.2.1 and Invariants on PTs.

□

Lemma 5.3.2 (Equivalence Preserved). $\forall st, st' \in \widehat{St}^{+}, i \in Idx$

If $st.\sigma_{\mathrm{HYP}}.curr = st'.\sigma_{\mathrm{HYP}}.curr = i$,
$st \underset{i}{\approx} st'$,
$st \xrightarrow{LoadState} st_1 \xrightarrow{GuestTrans(o)} st_2 \xrightarrow{SaveState} st_3$,
$st \xrightarrow{LoadState} st'_1 \xrightarrow{GuestTrans(o)} st'_2 \xrightarrow{SaveState} st'_3$,

Then $st_3 \underset{i}{\approx} st'_3$

Proof. The function $load\_state$ only depends on $regs_{\mathrm{virt}}.regs_{\mathrm{core}}$, $regs_{\mathrm{virt}}.regs_{\mathrm{bnk}}$ and of the mode and APSR bits of the CPSR of the current guest.

From Definition 5.3.6 of $\underset{i}{\approx}$ and Definition 5.3.5 of $same\_user\_regs\_hyp$, we know that all these registers are equal in the two initial states. The transition $LoadState$ writes the result of $load\_state$ to the hardware core registers and hardware APSR ($st.\sigma_{\mathrm{HW}}.regs_{\mathrm{core}}$ and $st.\sigma_{\mathrm{HW}}.apsr$).

Therefore $same\_user\_regs\_hw(st_1, st'_1)$ holds. Moreover, as $LoadState$ does not modify the state of the guest, $st_1 \underset{i}{\approx} st'_1$ holds.

As $same\_user\_regs\_hw(st_1, st'_1)$ and $st_1 \underset{i}{\approx} st'_1$ hold, by Axioms 3.5.2 and 3.5.1:

$$same\_map(st_2.mem, st'_2.mem, st_2.sthw.base_{\mathrm{PT}}, st'_2.sthw.base_{\mathrm{PT}})$$

As the states st_2 and st'_2 are well-formed, the PT of their base pointer register are equal to the SPT of their current guest, therefore:

$$same\_map(st_2.mem, st'_2.mem, st_2.sthyp.vcpus(i).base_{\mathrm{SPT}}, st'_2.sthyp.vcpus(i).base_{\mathrm{SPT}})$$

$same\_user\_region(st_2, st'_2)$ obviously holds, and the virtualized registers of the guest are not modified, therefore $same\_user\_regs\_hyp(st_2, st'_2)$ holds, meaning that $st_2 \underset{i}{\approx} st'_2$.

Furthermore, by Axioms 3.5.2 and 3.5.1, $same\_user\_regs\_hw(st_2, st'_2)$ holds. Same reasoning as for $load\_state$ function ensure that $save\_state$ copies the hardware registers into the guest virtualized registers in such a way that $same\_user\_regs\_hyp(st_3, st'_3)$.

Finally, $save\_state$ does not modify memory (see Section 3.5.2), therefore $st_3 \underset{i}{\approx} st'_3$.

□

Lemma 5.3.3 (Functional Relation). $\forall \sigma_G \in \widehat{St}^{+}_{G}$,

If $(\sigma_G, o) \underset{run}{\approx} (regs_{abs1}, priv_1, send_1) \wedge$
$(\sigma_G, o) \underset{run}{\approx} (regs_{abs2}, priv_2, send_2)$

Then $(regs_{abs1}, priv_1, send_1) = (regs_{abs2}, priv_2, send_2)$

Proof. By applying Lemma 5.3.1 twice and Lemma 5.3.2. □

From Lemma 5.3.3, we deduce that relation run is a function on $\widehat{St}_G^+$, we note $run : (\widehat{St}_G^+ \times Oracle) \to (Regs_{abs} \times Seg \times (Idx \to Seg))$.

Exit Similarly, we define $run\_exit : (\widehat{St}_G^+ \times Oracle) \to Exit\_code$. We proceed the same way as for the run relation. We define run_exit as a relation with [1] and [2] similar as for the run relation, and [3] specifying that the exit code corresponds to the exception-related information contained in the $regs_{\text{mmu}}$, $regs_{\text{gic}}$, mode and the exception vector obtained after a concrete transition. In particular, the exit code gives information about the exception raised at the end of the guest run, thus allowing to decide which handler is to be called. A similar reasoning as for the previous relation shows that this is a function.

We can now define the first part of the guest transition $guest\_run : (\widehat{St}_G^+ \times Oracle) \to St_G \times Exit\_code$.

Definition 5.3.7 (Guest Run). $\forall \sigma_G \in \widehat{St}_G^+, o \in Oracle$,
Let $(aregs, priv, send) = run(\sigma_G, o)$,
$exit\_code = run\_exit(\sigma_G, o)$

$$guest\_run(\sigma_G, o) = (aregs, priv, send, \sigma_G.rcv, exit\_code)$$

Guest Synchronize

The second step of the guest transition reports the changes of the send segments of the current guest to the corresponding receive segments of the other guests. In other terms, the receive segments of the other guests are synchronized with the send segment of the current guest. The synchronization of segment seg_1 with seg_2, i.e. updating all the values of seg_1 with those of seg_2 without changing its tags, is denoted by $seg_1 \xleftarrow{VAL} seg_2$.

Definition 5.3.8 (Synchronization). $\forall guests \in (Idx \to St_G), \sigma_G \in St_G, i \in Idx$,
Let $guests'(i).\sigma_G = \sigma_G$,
$\forall j \neq i, guests'(j) = guests(j)[rcv \xleftarrow{VAL} guests(i).send]$,

$synch(guests, \sigma_G, i) = guests'$

Whole Transition

The abstract guest transition is the sequence of the $guest\_run$ applied on the current guest, and of the synchronization of the current guest on the other guests.

Definition 5.3.9 (Guest Transition). $\forall st^\# \in \widehat{St}^{\#+}, o \in Oracle$,
Let $curr = st^\#.curr$
$\sigma_G = guest\_run(st^\#.guests(st^\#.curr), o)$
$guests_1 = synch(guests, \sigma_G, st^\#.curr)$

If $st_1^\# = st^\#[guests \leftarrow guests_1]$

Then $st^\# \xrightarrow{GuestTrans(o)} st_1^\#$

In this Section, we have defined the abstract run with help of some concrete transitions ($LoadState$, $Guest$ and $SaveState$). Yet we do not have proved its commutation with these

concrete transitions. Basically our definition gives us the commutation of the registers, of the send segments, and of the private segment of the current guest. We have two properties left to prove: the commutation over the receive segments of the current guest and the commutation over the state of the other guests.

5.3.3 Hypervisor Transition

We distinguish four types of *hypervisor transitions*, depending on their impact on the observable state: the Memory Management, the Schedule, the Nop and the Register Modification transitions. Each transition corresponds to one or several groups of the hypervisor concrete transitions presented in Section 3.5.3.

Memory Management

The *memory management* transition captures the effects of the concrete transitions concerning the memory management virtualization. These concrete transitions have an impact on the registers and on memory, but only on the part of memory which defines the SPT. In particular, they do not change the value of memory cells but only the active SPT. It means that the impact of the mem transition on the segments is only on their tag.

For this transition, we consider a function $mem\_op : St_G \times Exit\_code \to St_G$. This function first analyses the exit code and the registers, then goes to the appropriate handler. Instead of describing one transition per actual operation (map, $unmap$ etc...), we gather these transitions into one, which is precise enough to prove our properties. We detail the function mem_op in Section 5.5, as we need every detail to show the commutation with concrete transitions.

Definition 5.3.10 (Memory Management Transition)**.**

Let $curr = st^{\#}.curr$
$\sigma_{G_{curr}} = st^{\#}.guests(curr)$

If $decode(st^{\#}.aregs, exit\_code) \in \{pf, switch, flush, flushall, enable, disable\}$
$st_1^{\#} = st^{\#}[guests(curr) \leftarrow mem\_op(\sigma_{G_{curr}}, exit\_code)]$

Then $st^{\#} \xrightarrow{Mem} st_1^{\#}$

Schedule

The scheduling transition corresponds to the concrete scheduling transition. As explained in Section 5.3.1 uses the oracle to decide which guest to schedule.

Definition 5.3.11 (Schedule Transition)**.**

If $decode(st^{\#}.aregs, exit\_code) = schedule$
$st_1^{\#} = st^{\#}[curr \leftarrow nxt]$

Then $st^{\#} \xrightarrow{Schedule(nxt)} st_1^{\#}$

Nop

The *nop* transition is the abstraction of all the concrete transitions which do not have any observable impact on the abstract state. It abstracts all the IRQ transitions, indeed, all these transitions only impact the GIC which we do not represent in the abstract model.

Definition 5.3.12 (Nop Transition)**.**
If $decode(st^{\#}.aregs, exit\_code) = nop$
Then $st^{\#} \xrightarrow{Nop} st^{\#}$

Registers Modification

The *ModRegs* transition is the abstraction of the concrete injection transitions and of the access to privileged register transition. The particularity of these transitions is that their only observable impact is on registers.

Definition 5.3.13 (Modify Registers)**.**
Let $curr = st^{\#}.curr$
If $decode(st^{\#}.aregs, exit\_code) = mod\_regs$
$st_1^{\#} = st^{\#}.guests(curr)[aregs \leftarrow mod\_regs(aregs, exit\_code)]$
Then $st^{\#} \xrightarrow{ModRegs} st_1^{\#}$

5.3.4 Restore Transition

The view of the concrete **restore transition** does either nothing (in case just the PL is changed) or injects an IRQ into the guest, which only impacts the registers.

Definition 5.3.14 (Restore Transition)**.** The restore transition is a nop or uses the oracle to inject irqs if any.
Let $curr = st^{\#}.curr$
If $oirq \neq None$
Then $st_1^{\#} = st^{\#}[guests(curr).aregs \leftarrow oirq]$
$st^{\#} \xrightarrow{Restore(oirq)} st_1^{\#}$
Else $st^{\#} \xrightarrow{Restore(oirq)} st^{\#}$

5.3.5 Abstract Transition

Now that we have formally defined the restore transition, the guest transition and the hypervisor transition, we can express formally the Abstract Transition.

Definition 5.3.15 (Abstract Transition)**.** $\forall st^{\#}, st_1^{\#} \in \widehat{St}^{\#+}, o \in Oracle$,
$st^{\#} \xrightarrow{o} st_1^{\#} = st^{\#} \xrightarrow{Restore(o)} \xrightarrow{GuestTrans(o)} \xrightarrow{HypTrans(o)} st_1^{\#}$

5.4 Security properties

Guests may interfere with each other (e.g. through shared memory), so we cannot prove non-interference. Instead we prove an *isolation* property on some resources of the guests, i.e. we prove their integrity and their confidentiality. We have compared our property of isolation to non-interference in Section 1.2.

The resources on which we prove isolation are the registers and the memory segments. Below, we detail the properties on segments, as our main focus is the memory isolation. We express the properties on one transition step. More exactly, to state a property for one guest, we confine the effects that the execution of another guest can have on the former. Thus, as our system is sequential, we consider a transition where the former

guest does not run. We prove the extension of these properties to any sequence of transitions where a guest does not run. The proof sketch of integrity shows the simplicity with which we can bound the effects of a transition in our model. The proof of confidentiality is done in a similar way.

5.4.1 Integrity

Integrity for a guest i means that if another guest j runs, then the private segment and the send segments of i are not modified, and only its j^{th} receive segment might have changed.

Theorem 5.4.1 (Integrity). $\forall i, j$ such that $i \neq j, \forall guests, guests' \in (Idx \to St_G)$

If $\quad \langle j, guests\rangle \to \langle j', guests'\rangle$

Then $\quad guests'(i).priv = guests(i).priv$
$\forall k, guests'(i).send(k) = guests(i).send(k)$
$\forall k \neq j, guests(i)'.rcv(k) = guests(i).rcv(k)$

Proof. We consider a transition from the state $\langle j, guests\rangle$. From Definition 5.3.15, the first part of the transition is the restore transition, which only changes the registers of the running guest. So the segments of all the guests stay unchanged, in particular those of guest i. The current guest index is not modified, it is still j.

The second part of the transition is the guest transition. From the definition of the guest transition (Definition 5.3.9) we know that the current guest index is not changed, and that the state of guest i after the transition is such that only its j^{th} receive segment is modified. Hence the three following facts are verified:

1. $guests'(i).priv = guests(i).priv$
2. $\forall k, guests'(i).send(k) = guests(i).send(k)$
3. $\forall k \neq j, guests'(i).rcv(k) = guests(i).rcv(k)$.

The third part of the transition is the hypervisor transition. None of the four hypervisor transitions changes the state of guest i. Therefore integrity is verified for any transition. □

5.4.2 Confidentiality

To express confidentiality properties, we compare one step of execution from two states which differ only on some resources of guest i (Definitions 5.4.1 and 5.4.2). If the guest j which runs in this step has no authorization to access these resources, then the two states resulting from the transition are equal except on guest i.

We present two confidentiality properties, one for private segment, which ensure that only one guest can access its private segment, and one for shared segments, stating that only the two guests sharing a segment can access it.

For each property we define a notion of similarity.

Definition 5.4.1 (i-Similarity). For all $i \in Idx$, $\forall guests, guests' \in (Idx \to St_G)$, the elements $guests$ and $guests'$ are i-similar, noted $guests \underset{i}{\sim} guests'$ iff they only differ on the private region of guest i:

$$guests' \underset{i}{\sim} guests \Leftrightarrow \begin{array}{l} \forall l \neq i, guests'(l) = guests(l) \\ \forall l, guests'(i).send(l) = guests(i).send(l) \\ \forall l, guests'(i).rcv(l) = guests(i).rcv(l) \end{array}$$

Theorem 5.4.2 (Confidentiality Private Segment). $\forall i, j$ such that $i \neq j, \forall guests_1, guests_2 \in (Idx \to St_G)$ such that $guests_1 \underset{i}{\sim} guests_2$.

If $\langle j, guests_1 \rangle \xrightarrow{nxt,oirq,o} \langle j', guests_1' \rangle$

Then $\langle j, guests_2 \rangle \xrightarrow{nxt,oirq,o} \langle j', guests_2' \rangle \wedge guests_2' \underset{i}{\sim} guests_1'$

The proof of Theorem 5.4.2 is almost the same as the proof of the next theorem (Theorem 5.4.3).

Definition 5.4.2 (i,k-Similarity). For all $i \in Idx$, $\forall guests, guests' \in (Idx \to St_G)$, the elements $guests$ and $guests'$ are i,k-similar, noted $guests \underset{i,k}{\sim} guests'$ iff they only differ on the private region of guest i, and on the shared regions between i and k:
$\forall guests, guests' \in (Idx \to St_G)$,

$$guests' \underset{i,k}{\sim} guests \Leftrightarrow \begin{array}{l} \forall l \neq i, guests'(l) = guests(l) \\ \forall l \neq k, guests'(i).send(l) = guests(i).send(l) \\ \forall l \neq k, guests'(i).rcv(l) = guests(i).rcv(l) \end{array}$$

Theorem 5.4.3 (Confidentiality Shared Segments). $\forall i, j, k \in Idx$ such that $i \neq j$ and $k \neq j$, $\forall guests_1, guests_2 \in (Idx \to St_G)$ such that $guests_1 \underset{i,k}{\sim} guests_2$.

If $\langle j, guests_1 \rangle \xrightarrow{nxt,oirq,o} \langle j', guests_1' \rangle$

Then $\langle j, guests_2 \rangle \xrightarrow{nxt,oirq,o} \langle j', guests_2' \rangle \wedge guests_2' \underset{i,k}{\sim} guests_1'$

Proof. We consider a transition from the two states $\langle j, guests_1 \rangle$ and $\langle j, guests_2 \rangle$. We let $l \in Idx$. We reason on each part of the transition (Definition 5.3.15).

The restore transition either does nothing or changes the registers of the running guest, depending on the oracle. We consider two transitions with the same oracle, hence the relation $\underset{i,k}{\sim}$ is maintained.

The second part of the transition is the guest transition, for which we distinguish two cases:

$\boxed{l = j}$ by Definition 5.3.7 of $guest\_run$, the new value of the j^{th} guest only depends on the oracle o and of the state of guest j. Therefore the j^{th} guest is equals for the two states.

$\boxed{l \neq j}$ From the definition of the synchronization (Definition 5.3.8) we know that the state of guest l after the transition is such that only its j^{th} receive segment is modified, with the values of the send segment of j, which, we have just proved, is equal for the two states. Thus the relation $\underset{i,k}{\sim}$ is maintained.

The third part of the transition is the hypervisor transition. This transition only depends on the state of guest j, which is the same in the two states considered, and of the oracle, therefore $\underset{i,k}{\sim}$ is maintained. Thus confidentiality is verified for any transition. □

In Theorems 5.4.2 and 5.4.3, we consider assume that when a guest does not run, its memory does not interfere with oracles, i.e. that behavior of the scheduler and of the

interrupt management do not depend on the memory of a guest, when this guest is not the current guest.

The arrival of IRQs is an external event so it is obviously not linked to the configuration of memory. For the two oracles introduced in this chapter, our assumption can be proved by showing that two concrete states corresponding to two similar abstract states would schedule the same guest and inject the same IRQ. We set these properties aside in order to focus on memory isolation.

5.5 Refinement

Finally, we present our commutation lemmas and their proof. As illustrated in Figure 5.5, we want to show that, if there exist a transition from concrete state st to st', and if there is a view of st. Then there is a view of st' and this view is equal to the result of the abstract transition from $view(st)$.

We prove the commutation for the guest transition and for the memory management transitions. These proofs rely on the invariants described in Chapter 4, and on the resulting effects.

5.5.1 Guest Transition

The guest transition does not modify the field $curr$ of a state. We only express the commutation on the guests.

Lemma 5.5.1 (Guest Transition Commutation)**.**

Let $st \in St$ such that $wf^+(st)$,

If $st \xrightarrow{LoadState} \xrightarrow{GuestTrans(o)} \xrightarrow{SaveState} st'$

$view(st) = st^\#$

$st^\# \xrightarrow{GuestTrans^\#(o)} st_1^\#$

Then $view(st') = st_1^\#$

Proof. For clarity we introduce the following notations:

$$
\begin{aligned}
st^{\#\prime} &= view(st') \\
guests' &= st^{\#\prime}.guests \\
guests &= st^\#.guests \\
guests_1 &= st^\#.guests_1 \\
mem &= st.\sigma_{\mathrm{HW}}.mem \\
mem' &= st'.\sigma'_{\mathrm{HW}}.mem \\
vcpus &= st.\sigma_{\mathrm{HYP}}.vcpus \\
vcpus &= st.\sigma_{\mathrm{HYP}}.vcpus, \text{ also equal to } st'.\sigma'_{\mathrm{HYP}}.vcpus, \text{ by Definition 3.5.1} \\
base_{\mathrm{SPT}} &= vcpus(i).base_{\mathrm{SPT}}
\end{aligned}
$$

Suppose that $st^{\#\prime} \neq st_0^\#1$, it means that one of their field differ. From the definitions of the concrete and abstract guest transition, and from the definition of the view, we know that $st^{\#\prime}.curr = st_1^\#.curr$, and we denote by i such a current guest.

Therefore $st^{\#\prime} \neq st_1^\#$ means that there exists $j \in Idx$ such that $guests'(j) \neq guests_1(j)$.

Case $i = j$. In this case, by Definition 5.3.9 and 5.3.8:

$$guests_1(i).\sigma_G = guest\_run(guests(i), o)$$

By definition of $guest\_run$ and of the run relation (Definitions 5.3.7 and 5.3.3), we already have the commutation of the registers, private segment and send segment. We prove it for the receive segments.

Suppose then that $guests'(i).rcv \neq guests_1(i).rcv$ (1), by Definition 5.3.7, we know that:

$$guests_1(i).rcv = guests(i).rcv$$

Therefore, from (1), we obtain:

$$guests'(i).rcv \neq guests(i).rcv$$

We write rcv' for $guests'(i).rcv$ and rcv for $guests(i).rcv$. There exists $k \in Idx, pa \in Addr$ such that $rcv'(j)(pa) \neq rcv(j)(pa)$ (2).

Case $pa \notin shared(k, i)$. From Definition 5.2.6:

$$rcv'(j)(pa) = rcv(j)(pa) = none$$

which contradicts (2). Qed.

Case $pa \in shared(k, i)$. From Definition 5.2.6:

$$rcv'(j)(pa) = \langle mem'(pa), tags(mem', vcpus', j, \boldsymbol{pa})\rangle$$
$$\wedge\, rcv(j)(pa) = \langle mem(pa), tags(mem, vcpus, j, \boldsymbol{pa})\rangle$$

From Lemma 4.3.7, we know that $mem'(pa) = mem(pa)$. It means that the tags are different.

By Lemma 5.2.1:

$$rcv'(j)(pa).tags = \{\langle va, r\rangle | pt(mem', base_{\text{SPT}})(va) = \langle pa, r\rangle\}$$
$$\wedge\, rcv'(j)(pa).tags = \{\langle va, r\rangle | pt(mem, base_{\text{SPT}})(va) = \langle pa, r\rangle\}$$

From Lemma 4.3.8, we have that

$$\forall va, pt(mem', base_{\text{SPT}})(va) = pt(mem, base_{\text{SPT}})(va)$$

It means that $rcv'(j)(pa).tags = rcv(j)(pa).tags$. Qed.

Case $i \neq j$. Let $\sigma_G = guest\_run(guests(i), o)$. By Definition 5.3.9 and 5.3.8:

$$guests_1(j) = guests(j)[rcv \xleftarrow{VAL} \sigma_G.send] \text{ (3)}$$

It means that $guests_1(j)$ and $guests(j)$ only differ on the receive segment. Similar reasoning as previously show that $guests(j)$ and $guests'(j)$ are equal on registers, send and private segments, because the component needed for their view are not modified by the concrete guest transition. Thus that $guests_1(j)$ and $guests(j)$ are equals on registers, send and private segments.

We analyze the case where $guests'(i).rcv \neq guests_1(i).rcv$. There exists $k \in Idx$ and $pa \in Addr$ such that $rcv'(j)(pa) \neq rcv_1(j)(pa)$ (4).

Case $pa \notin shared(k, i)$. From Definition 5.2.6:

$$rcv'(j)(pa) = rcv(j)(pa) = none$$

From (3), $rcv(j)(pa) = none$ implies that $rcv_1(j)(pa) = none$. which contradicts (4). Qed.

Case $pa \in shared(k, i)$. From Definition 5.2.6:

$$rcv'(j)(pa) = \langle mem'(pa), tags(mem', vcpus', j, \boldsymbol{pa})\rangle$$
$$rcv(j)(pa) = \langle mem(pa), tags(mem, vcpus, j, \boldsymbol{pa})\rangle$$

From (4), we know that $rcv_1(j)(pa).tags = rcv(j)(pa).tags$. Same reasoning as before, using the definition of tags and Lemma 4.3.8 leads to the conclusion that:

$$rcv'(j)(pa).tags = rcv_1(j)(pa)$$

Suppose then that $mem'(pa) \neq rcv_1(j)(pa).byte$ (5). From (4), we know that:

$$rcv_1(j)(pa).byte = guest\_run(guests(i), o).send(j)(pa).byte$$

By definition of $guest\_run$ and of the run relation (Definitions 5.3.7 and 5.3.3), we know that:

$$guest\_run(guests(i), o).send(j)(pa).byte = view_{send}(mem', perm, vcpus, i)(j)(pa)$$

which is equal to $mem'(pa)$ by Definition 5.2.5. Meaning that $mem'(pa) = rcv_1(j)(pa).byte$, which contradicts (5). Qed.

□

5.5.2 Memory Transitions

There are six hypervisor transitions. The page fault with MMU and without MMU call the map operation. The flush transition calls the $unmap$ operation. The switch, flush all and enable/disable MMU transitions call the $unmap\_all$ operation. The mem_op operation that we have presented in the the abstract memory management transition would call one of these operation, depending on the exit arguments of the guest transition. We prove the commutation of the concrete and abstract map, $unmap$ and $unmap\_all$ operations, in order to prove the commutation of all the memory management related transitions.

Map

We want to prove the correspondence between the concrete and an abstract map. The abstract map, noted $map^{\#}$, add tags containing a virtual address within the virtual page to map, to cells located at physical addresses in the physical page to map. It has the following type $map^{\#} : (St_G \times Addr \times Addr \times Rights) \rightarrow St_G$. It also removes the tags containing a virtual address located in the page to be mapped. Indeed, this captures the side effect of the concrete map, which, by mapping a virtual address va to a physical page, also unmap all the older occurrences of va mappings.

The addition of new mappings is made at the granularity of a page, we map the whole page at the given addresses. Therefore our specification of $map^{\#}$ only makes sense for

addresses aligned on the size of the page, and the commutation needs to be proved only on addresses aligned to the size of a page. We denote by $rm\_page : (\{Tag\} \times Addr_{pg}) \rightarrow \{Tag\}$ the function that removes the tags containing an address in a page at the address given in argument.

Definition 5.5.1 (Remove Tags)**.** Let $tags \in Tag, va_{pg} \in Addr$,

$$rm\_page(tags, va_{pg}) = tags \backslash \{tag | tag.va \in [va_{pg}, va_{pg} + page\_size[\ \}$$

We denote by $add\_page : (\{Tag\} \times Addr_{pg} \times Addr \times Rights) \rightarrow \{Tag\}$ the function which add a tag containing an address in a page at the address given in argument.

Definition 5.5.2 (Add Tags)**.** $\forall tags \in Tag, \forall va_{pg} \in Addr_{pg}, \forall pa \in Addr, \forall r \in Rights$, let $off \in Uint$ such that $pa = idx_1 \oplus idx_2 \oplus off$,

$$add\_page(tags, va_{pg}, pa, r) = tags' \cup (va_{pg} + off, r)$$

We decompose the proof of commutation in four, one for each component of the abstract state. We define the projection of $map^{\#}$on each component of the system. Formally, $\forall st^{\#} \in St^{\#}, \forall va_{pg} \in Addr$:

$$map^{\#}(\sigma_G, va_{pg}, pa_{pg}, r) = \begin{cases} st^{\#}.regs \\ map^{\#}_{priv}(st^{\#}.priv, va_{pg}, pa_{pg}, r) \\ map^{\#}_{send}(st^{\#}.send, va_{pg}, pa_{pg}, r) \\ map^{\#}_{rcv}(st^{\#}.rcv, va_{pg}, pa_{pg}, r) \end{cases}$$

The commutation of the map function on the registers is easy, because the map function does not modify the registers.

We specify below the function $map^{\#}_{priv}$:

Definition 5.5.3 (Map Priv)**.** $\forall seg \in Seg, va_{pg} \in Addr_{pg}, a \in Addr$,

$map^{\#}_{priv}(seg, va_{pg}, pa_{pg}, r)(a) =$ If $seg(a) \in None$
Then **return** ***none***
Else Let $b = seg(a).byte$
$tags = rm\_page(seg(a).tags, va_{pg})$
If $a \in [pa_{pg}, size_{page}[$
Then Let $tags' = add\_page(tags, va_{pg}, pa, r)$
return $\langle b, tags' \rangle$
Else **return** $\langle b, tags \rangle$

Lemma 5.5.2 (Commutation of Map - Private Segment)**.** If we unmap va_{pg} in a concrete state, then the view of the private segment of this state is equal to unmapping va_{pg} in the view of the initial concrete state:

Let $st \in St$ such that $wf(st)$,
$i \in Idx$,
$va_{pg} \in Addr_{pg}$ such that $va_{pg} \notin \sigma_{\mathrm{HYP}}.vcpus(i).frange$,
$st' \in St$,
$seg \in Seg$,

If $st' = map(st, i, va_{pg}, pa_{pg}, r)$
$seg = view_{priv}(st.\sigma_{\mathrm{HW}}.mem, st.\sigma_{\mathrm{HYP}}.vcpus, i)$

Then $view_{priv}(st'.\sigma_{\mathrm{HW}}.mem, perm, st'.\sigma_{\mathrm{HYP}}.vcpus, i) = map^{\#}_{priv}(seg, va_{pg})$

Proof. For clarity we introduce the following notations:

$$\begin{array}{ll} seg_1 & = map^{\#}_{priv}(seg, va_{pg}) \\ seg' & = view_{priv}(st'.\sigma_{\mathrm{HW}}.mem, st'.\sigma_{\mathrm{HYP}}.vcpus, i) \\ mem & = st.\sigma_{\mathrm{HW}}.mem \\ mem' & = st'.\sigma'_{\mathrm{HW}}.mem \\ vcpus & = st.\sigma_{\mathrm{HYP}}.vcpus \\ vcpus' & = st'.\sigma_{\mathrm{HYP}}.vcpus \end{array}$$

Assume that $seg' \neq seg_1$. Then there exists a physical address $pa \in Addr$ such that $seg'(pa) \neq seg_1(pa)$, **we denote by $\mathcal{H}$ this hypothesis.**

By Definition 4.2.2 (Map Page), for all $j \neq i$, $vcpus(j) = vcpus(i)$ and $vcpus(i)$ is only modified on is $alloc$ field. In particular, $vcpus(i).base_{\mathrm{SPT}} = vcpus'(i).base_{\mathrm{SPT}}$, which we write $base_{\mathrm{SPT}}$ for short.

Case $pa \notin perm.priv(i)$. By Definition 5.2.4, $seg(pa) = Oob$ and $seg'(pa) = Oob$. From Definition 5.5.4, $seg(pa) = Oob$ implies that $seg_1(pa) = Oob$.
Thus $seg'(pa) = seg_1(pa)$, which contradicts $\mathcal{H}$, Qed.

Case $pa \in perm.priv(i)$. By Definition 5.2.4:

$$seg'(pa) = \langle mem'(pa), tags(mem', vcpus', i, pa)\rangle \quad (1)$$
$$\wedge seg(pa) = \langle mem(pa), tags(mem, vcpus, i, pa)\rangle \quad (2)$$

Yet from Lemma 4.3.2 (Map Unchanged), we know that $mem(pa) = mem'(pa)$, i.e. that:

$$seg'(pa).byte = seg_1(pa).byte$$

As $seg'(pa) \neq seg_1(pa)$ ($\mathcal{H}$), we have:

$$seg'(pa).tags \neq seg_1(pa).tags$$

As the two sets are not equals, there exist an element e which is in the first set but not in the second, or reciprocally. We introduce the virtual address $v_e \in Addr$ and the right $r_e \in Rights$, such that $e = \langle v_e, r_e\rangle$.

Case $e \in seg'(pa).tags \wedge e \notin seg_1(pa).tags$. We know by (1) that:

$$seg'(pa).tags = tags(mem', vcpus', i, pa)$$

it means that:

$$seg'(pa).tags = \{\langle va, r\rangle | pt(mem', base_{\mathrm{SPT}})(va) = \langle pa, r\rangle\}$$

from Lemma 5.2.1. Therefore, as $\langle v_e, r_e\rangle \in seg'(pa).tags$:

$$pt(mem', base_{\mathrm{SPT}})(v_e) = \langle pa, r_e\rangle \quad (4)$$

Case $v_e \in [va_{pg}, va_{pg} + page\_size[$. Then from Lemma 4.3.1:

$$\exists off < size_{page}, v_e = va_{pg} + off, pt(mem', base_{\mathrm{SPT}})(v_e) = (pa_{pg} + off, r)$$

From (4), we obtain that $pa_{pg}+off = pa$ and $r = r_e$. It means that $pa \in [pa_{pg}, size_{page}[$, thus, let $tags = rm\_page(seg(pa).tags, va_{pg})$, from Definition 5.5.3:

$$seg_1(pa).tags = add\_page(tags, va_{pg}, pa, r_e)$$

From Definition 5.5.2:

$$add\_page(tags, va_{pg}, pa, r_e) = tags \cup (v_e, r_e)$$

Therefore $(v_e, r_e) \in seg_1(pa).tags$, which contradicts the hypothesis of the case. Qed.

Case $v_e \notin [va_{pg}, va_{pg} + page\_size[$. Then from Lemma 4.3.1:

$$pt(mem, base_{\text{SPT}})(v_e) = pt(mem', base_{\text{SPT}})(v_e) = \langle pa, r \rangle$$

As we know from (2) that:

$$seg(pa).tags = tags(mem, vcpus, i, pa)$$

we deduce from the specification of tags (Lemma 5.2.1) that:

$$seg(pa).tags = \{\langle va, r \rangle \mid pt(mem, base_{\text{SPT}})(va) = \langle pa, r \rangle\}$$

In particular it means that:

$$\langle v_e, r_e \rangle \in seg(pa).tags \quad (5)$$

Case $pa \notin [pa_{pg}, size_{page}[$ From Definition 5.5.3:

$$seg_1(pa) = \langle seg(a).byte, rm\_page(seg(a).tags, va_{pg}) \rangle$$

Therefore, from Definition 5.5.1, we know that:

$$rm\_page(seg(a).tags, va_{pg}) = seg(a).tags \backslash \{tag \mid tag.va \in [va_{pg}, va_{pg}+page\_size]\}$$

As $v_e \notin [va_{pg}, va_{pg} + page\_size]$:

$$\langle v_e, r_e \rangle \in seg(a).tags \Leftrightarrow \langle v_e, r_e \rangle \in seg_1(a).tags$$

Yet we are in the case where $e \notin seg_1(pa).tags$ and we know from (5) that $e \in seg(pa).tags$, which is contradictory. Qed.

Case $pa \in [pa_{pg}, size_{page}[$ Let $tags = rm\_page(seg(pa).tags, va_{pg})$. The same reasoning as for the previous case leads to $e \in tags$.
From Definition 5.5.3:

$$seg_1(pa) = \langle seg(pa).byte, add\_page(tags, va_{pg}, pa, r) \rangle$$

as add_page only *adds* tags, $e \in seg_1(pa).tags$, which is contradictory with our case. Qed.

Case $e \notin seg'(pa).tags \wedge e \in seg_1(pa).tags$. As $e \notin seg'(pa).tags$, from the specification of tags (Lemma 5.2.1) it means that pa is not mapped by v_e with rights r_e in the current

page table of guest i in the new mem':

$$pt(mem', spt)(v_e) \neq \langle pa, r_e \rangle \text{ (6)}$$

Case $v_e \in [va_{pg}, va_{pg} + size_{page}[$. From Lemma 4.3.1,

$$\exists off < size_{page}, v_e = va_{pg} + off, pt(mem', base_{\text{SPT}})(v_e) = (pa_{pg} + off, r)$$

From (6), we obtain that $pa_{pg} + off \neq pa$ or $r \neq r_e$ (7).

Case $pa \notin [pa_{pg}, size_{page}[$ We know that $seg_1.tags = rm\_page(seg(a).tags, va)$. From Definition 5.5.1:

$$rm\_page(seg(a).tags, va_{pg}) = seg(a).tags \backslash \{tag | tag.va \in [va_{pg}, va_{pg} + page\_size]\}$$

As $v_e \in [va_{pg}, va_{pg} + size_{page}]$, $\langle v_e, r_e \rangle \notin seg_1(a).tags$. Qed.

Case $pa \in [pa_{pg}, size_{page}[$ Let $tags = rm\_page(seg(pa).tags, va_{pg})$ The same reasoning as for the previous case leads to $e \notin tags$.
From Definition 5.5.3:

$$seg_1(pa).tags = add\_page(tags, va_{pg}, pa, r)$$

Let $off_1 < size_{page}$ such that $pa_{pg} + off_1 = pa$, from Definition 5.5.2:

$$add\_page(tags, va_{pg}, pa, r) = tags \cup (va + off_1, r)$$

From (7), we deduce that $(va + off_1, r) \neq (v_e, r_e)$. Therefore $(v_e, r_e) \notin seg_1(pa).tags$. Qed.

Case $v_e \notin [va_{pg}, va_{pg} + size_{page}[$. From Lemma 4.3.1, we know that the mappings from v_e are equivalents in the initial and the new memory. Thus (6) implies that

$$pt(mem, spt)(v_e) \neq \langle pa, r_e \rangle$$

Therefore $e \notin seg(pa).tags$, from Lemma 5.2.1.

Case $pa \notin [pa_{pg}, size_{page}[$ We know from Definition 5.5.1 that

$$seg_1.tags = rm\_page(seg(a).tags, va_{pg})$$

Yet $rm\_page(seg(a).tags, va_{pg})$ only removes tags, so $e \notin seg(pa).tags$ implies $e \notin seg_1(pa).tags$, which contradicts our case. Qed.

Case $pa \in [pa_{pg}, size_{page}[$ Let $tags = rm\_page(seg(pa).tags, va_{pg})$. The same reasoning as for the previous case leads to $e \notin tags$.
From Definition 5.5.3:

$$seg_1(pa).tags = add\_page(tags, va_{pg}, pa, r)$$

From Definition 5.5.2, let $off < size_{page}$ such that $pa_{pg} + off = pa$

$$add\_page(tags, va_{pg}, pa, r) = tags \cup (va + off, r)$$

As $e \notin tags$ and $e \in add\_page(tags, va_{pg}, pa, r)$, $v_e = va + off$. Yet $va + off \in [va_{pg}, va_{pg} + size_{page}[$, which contradicts the case. Qed.

□

Unmap

The abstract $unmap$, noted $unmap^{\#}$, removes the tags containing an address located in the same page as va from all the segments' cells. It has the following type

$$unmap^{\#} : (\sigma_G \times Addr) \to \sigma_G$$

Similarly to the $map^{\#}$ function, our specification of $unmap^{\#}$ only makes sens for virtual addresses aligned to the size of a page.

Similarly as for $map^{\#}$ we define the projection of $unmap^{\#}$on each component of the system. Formally, $\forall st^{\#} \in St^{\#}, \forall va_{pg} \in Addr$:

$$unmap^{\#}(\sigma_G, va_{pg}) = \begin{cases} st^{\#}.regs \\ unmap^{\#}_{priv}(st^{\#}.priv, va_{pg}) \\ unmap^{\#}_{send}(st^{\#}.send, va_{pg}) \\ unmap^{\#}_{rcv}(st^{\#}.rcv, va_{pg}) \end{cases}$$

We specify below the function $unmap^{\#}$ on the private segment. Again, $unmap$ does not modify registers, therefore the commutation for registers does not present any difficulties.

Definition 5.5.4 (Unmap Priv). $\forall seg \in Seg, va_{pg} \in Addr_{pg}, a \in Addr$,

$unmap^{\#}_{priv}(seg, va_{pg})(a) =$ If $seg(a) \in None$
Then **return** ***none***
Else Let $b = seg(a).byte$
$tags = rm\_page(seg(a).tags, va)$
return $\langle b, tags \rangle$

We have proved in Section 4.3.2, that under some requirements of well-formedness on the concrete state (invariant properties), the concrete $unmap$ function respects the Lemmas 4.3.3 and 4.3.4. We use it to prove the lemma of commutation of $unmap$ for the private segment, which we state below.

Lemma 5.5.3 (Commutation of Unmap - Private Segment). If we unmap va_{pg} in a concrete state, then the view of the private segment of this state is equal to unmapping va_{pg} in the view of the initial concrete state:

Let $st \in St$ such that $wf(st)$,
$i \in Idx$,
$va_{pg} \in Addr_{pg}$ such that $va_{pg} \notin \sigma_{\text{HYP}}.vcpus(i).frange$,
$st' \in St$,
$seg \in Seg$,

If $st' = unmap(st, i, va_{pg})$
$seg = view_{priv}(st.\sigma_{\text{HW}}.mem, st.\sigma_{\text{HYP}}.vcpus, i)$

Then $view_{priv}(st'.\sigma_{\text{HW}}.mem, perm, st'.\sigma_{\text{HYP}}.vcpus, i) = unmap^{\#}_{priv}(seg, va_{pg})$

Proof. For clarity we introduce the following notations:

$$\begin{array}{ll} seg_1 & = unmap^{\#}_{priv}(seg, va_{pg}) \\ seg' & = view_{priv}(st'.\sigma_{\text{HW}}.mem, st'.\sigma_{\text{HYP}}.vcpus, i) \\ mem & = \sigma_{\text{HW}}.mem \\ mem' & = \sigma'_{\text{HW}}.mem \\ vcpus & = st.\sigma_{\text{HYP}}.vcpus, \text{ also equal to } st'.\sigma'_{\text{HYP}}.vcpus, \text{ by definition} \end{array}$$

Assume that $seg' \neq seg_1$. Then there exists a physical address $pa \in Addr$ such that $seg'(pa) \neq seg_1(pa)$, we denote by $\mathcal{H}$ this hypothesis.

Case $pa \notin perm.priv(i)$. By Definition 5.2.4, $seg(pa) = Oob$ and $seg'(pa) = Oob$. From Definition 5.5.4, $seg(pa) = Oob$ implies that $seg_1(pa) = Oob$.
Thus $seg'(pa) = seg_1(pa)$, which contradicts $\mathcal{H}$, Qed.

Case $pa \in perm.priv(i)$. By Definition 5.2.4:

$$seg'(pa) = \langle mem'(pa), tags(mem', vcpus, i, pa)\rangle \text{ (1)}$$
$$seg(pa) = \langle mem(pa), tags(mem, vcpus, i, pa)\rangle \text{ (2)}$$

As $seg(pa) \in Cell$, from Definition 5.5.4:

$$seg_1(pa) = \langle seg(pa).byte, rm\_page(seg(pa).tags, va)\rangle \text{ (3)}$$

Yet from Lemma 4.3.4, we know that $mem(pa) = mem'(pa)$, i.e. that:

$$seg'(pa).byte = seg_1(pa).byte$$

As $seg'(pa) \neq seg_1(pa)$ ($\mathcal{H}$), we have $seg'(pa).tags \neq seg_1(pa).tags$.

As the two sets are not equals, there exist an element e which is in the first set but not in the second, or reciprocally. We introduce the virtual address $v_e \in Addr$ and the right $r_e \in Rights$, such that $e = \langle v_e, r_e\rangle$.

Case $e \in seg'(pa).tags \wedge e \notin seg_1(pa).tags$. We know by (1) that:

$$seg'(pa).tags = tags(mem', vcpus, i, pa)$$

From Lemma 5.2.1, it means that:

$$seg'(pa).tags = \{\langle va, r\rangle | pt(mem', spt)(va) = \langle pa, r\rangle\}$$

Therefore, as $\langle v_e, r_e\rangle \in seg'(pa).tags$:

$$pt(mem', spt)(v_e) = \langle pa, r_e\rangle \text{ (4)}$$

Case $v_e \in [va_{pg}, va_{pg} + page\_size[$. Then from Lemma 4.3.3:

$$pt(mem', spt)(v_e) = fault$$

Which contradicts (4). Qed.

Case $v_e \notin [va_{pg}, va_{pg} + page\_size[$. Then from Lemma 4.3.3:

$$pt(mem', spt)(va) = \langle a, r\rangle \Leftrightarrow pt(mem, spt)(va) = \langle a, r\rangle$$

As we know from (2) that $seg(pa).tags = tags(mem, vcpus, i, pa)$, we deduce from the Lemma 5.2.1 of the tags, that:

$$seg(pa).tags = \{\langle va, r\rangle \mid pt(mem, spt)(va) = \langle pa, r\rangle\}$$

In particular it means that:

$$\langle v_e, r_e\rangle \in seg(pa).tags \text{ (5)}$$

We know from (3) that $seg_1.tags = rm\_page(seg(pa).tags, va_{pg})$. From Definition 5.5.1, we know that:

$$rm\_page(seg(a).tags, va_{pg}) = seg(pa).tags \backslash \{tag \mid tag.va \in [va_{pg}, va_{pg} + size_{page}]\}$$

As $v_e \notin [va_{pg}, va_{pg} + size_{page}]$:

$$\langle v_e, r_e\rangle \in seg(pa).tags \Leftrightarrow \langle v_e, r_e\rangle \in seg_1(pa).tags$$

Yet we are in the case where $e \notin seg_1(pa).tags$ and we know from (5) that $e \in seg(pa).tags$, which is contradictory. Qed.

Case $e \notin seg'(pa).tags \wedge e \in seg_1(pa).tags$. As $e \notin seg'(pa).tags$, from Lemma 5.2.1, it means that pa is not mapped by v_e with rights r_e in the current page table of guest i in the new mem':

$$pt(mem', spt)(v_e) \neq \langle pa, r_e\rangle \text{(6)}$$

Case $v_e \in [va_{pg}, va_{pg} + page\_size[$. We know that $seg_1.tags = rm\_page(seg(pa).tags, va)$. From Definition 5.5.1:

$$rm\_page(seg(pa).tags, va_{pg}) = seg(pa).tags \backslash \{tag | tag.va \in [va_{pg}, va_{pg} + size_{page}]\}$$

As $v_e \in [va_{pg}, va_{pg} + size_{page}]$, $\langle v_e, r_e\rangle \notin seg_1(pa).tags$. Qed.

Case $v_e \notin [va_{pg}, va_{pg} + size_{page}[$. From Lemma 4.3.3, we know that the mappings from v_e are equivalents in the initial and the new memory. Thus (6) implies that $pt(mem, spt)(v_e) \neq \langle pa, r_e\rangle$. Therefore $e \notin seg(pa).tags$, from Lemma 5.2.1.
From Definition 5.5.1, we know that

$$seg_1.tags = rm\_page(seg(pa).tags, va_{pg})$$

Yet $rm\_page(seg(pa).tags, va_{pg})$ only removes tags, so $e \notin seg(pa).tags$ implies $e \notin seg_1(pa).tags$, which contradicts our case. Qed.

□

Unmap All

We proceed the same way as we did fo $unmap$ and map. We present the definition for the abstract $unmap\_all$ on private segment.

Definition 5.5.5 (Unmap All Priv). $\forall seg \in Seg, va_{pg} \in Addr_{pg}, a \in Addr,$

$unmap^{\#}_{priv}(seg, va_{pg})(a) =$ If $seg(a) \in None$
Then **return** ***none***
Else Let $b = seg(a).byte$
$tags = \{\}$
return $\langle b, tags \rangle$

This function is the most simple of the three presented, and its proof is also simpler. Indeed, Lemma 4.3.5 about the effects of the concrete $unmap\_all$ does not depend on the address considered: there is no more user mapping in the SPT. Given the Definition 5.2.4 of the private segment abstraction, the link between the concrete unmap, and this abstract unmap, where there is no more tags, is straightforward. Furthermore, although Lemma 4.3.5 is more difficult to prove than the map and $unmap$ counterparts, the commutation is a lot easier.

5.6 Impact of Optimizations on the Abstract Model

As illustrated by Figure 5.1, there are three components defining memory abstraction: the static permissions which define the domain of each segment, the user regions of memory which define the byte value in each memory cell, and the SPTs which define the tag mask of each segment.

In this section, we list three optimizations and generalizations and review their impact on the abstract model. This section aims at showing that our work is not limited to our particular hypervisor, but could be reused for hypervisors slightly different.

5.6.1 Several SPTs per Guest

When the guest switches of GPT, i.e. when it attempts to modify its pointer to GPT, it traps to the hypervisor, which emulate this operation by changing the current SPTs. As the hypervisor has only one slot per guest for holding the SPT, when the guest wants to switch to another SPT, the hypervisor has to flush the current SPT.

An optimization that we have not included in our model is the handling of several SPT slots per guest. This way, the hypervisor does not need to flush the SPT on every switch, it just uses another slot.

From the abstract model perspective, flushing the SPT means removing all the tags of the mask. If we have several SPT though, we must maintain in the abstract model several masks per guest. In this case, it is more convenient to separate completely the masks from the segments. Which means defining the segment as a function from addresses to bytes, and defining a mask as being a function from addresses to tags.

5.6.2 Allocator

We have described an allocator of pages of the $pool$, which is used in the map operation to allocate new second level PTs, and in the $unmap\_all$ to deallocate second level PTs. In the map operation, if there are no more free pages available in the allocator, the operation just fails. This could be optimized by returning a page to the pool when no more pages are available. In terms of observable behavior, it means that the map operation, besides adding some mappings, would also remove some mappings. Therefore, we would need to add information in the abstract domain in order to decide which mappings to remove.

Depending on the algorithm used to choose the page to evict, this modification may be important. Especially if the algorithm is purely based on the address of the pool physical pages, because it is related to PT structure, which we have totally abstracted. On the contrary, if the algorithm is based on the age of the allocated page (e.g. removing the oldest allocated page), we can easily keep track of this information in our model.

5.6.3 Dynamic Configuration

The two previous issues had an impact on SPTs, i.e. on the abstract model's *tags*. In this part we discuss the modification on the static configuration, i.e. on the *domain* of abstract segments.

Our abstraction rely on permissions defined by the hypervisor. In our case, these permissions are static, but in some other context, we may need permissions to be dynamic. For instance, a static configuration does not allow to launch new guests at runtime.

Rendering the global configuration dynamic would not modify the abstract state, but only add some transitions which would for example change some $None$ cells to real cells and vice versa.

If no permissions are defined, i.e. if the hypervisor allocates pages for guests, without following a global configuration, it is more difficult to design an abstract model which makes sense. For example, suppose we have two guests which do not need to share memory. The hypervisor makes sure that the SPTs of the two guests do not map the same addresses. In this case we can derive the permissions from the mappings presents in the SPTs, and present an abstract model where each guest has its own private segments.

In the case where guests can share some parts of the memory, we can still derive the permissions from the mappings presents in the SPTs, and simulate the system by an abstract system where addresses presents in several SPTs are in *shared* regions. However, the property of isolation as we have stated it cannot be proved in the general case, because a guest could map virtual addresses in the private region of another guest, meaning that the secret resource of a guest would be modified while the guest is not running. Therefore in case of sharing, the abstraction must be linked to some kind of policy which allows to define what should be shared or not.

Therefore, our model with static permissions is not that simplistic, and can be extended to a dynamic model without modifying too much the abstract model.

5.7 Key Points

- We have presented our abstract transition system, and the abstraction function.
- The abstract state is designed in a way such that some properties intrinsically hold.
- The transitions are deterministic.
- We have proved isolation on our abstract system.
- We have proved that the concrete system refines our abstract system, i.e. that the isolation is verified at concrete level.

Chapter 6

Benchmarks and Measurements

Contents

In this small chapter, we show benchmarks about the hypervisor we have studied, and we quantify our proof effort.

6.1 Benchmarks

The hypervisor that we have studied in this thesis was developed by the team Security in Telecommunications (SecT) at Technische Universität Berlin [Sec; Nor+15; Vet+15]. The code of the hypervisor is about 6000 sloc.

The benchmarks that we present below were performed by SecT. Note that the benchmarks are performed on a more recent version of the hypervisor than the one we worked on. Indeed, we had to consider a stable version in order to design our models and perform our proofs, and we did not take into account the optimizations made after March 2014. However there is no major change in the design, in particular, our low-level modeling of the SPT management is still valid for this new version.

The LMBench are low-level benchmarks developed for Linux, which compute the execution time of some common low-level operations, such as the time of memory copy/read/write, the cost of a page fault, the data movement through pipes and so on [Lmb]. Several benchmarks from LMbench were performed on a virtualized Linux v4.2 running alone on the hypervisor. We present the results in Table 6.1, and compute the overhead compared to a native execution of Linux. The simple syscall (resp. read, write) corresponds to the LMBench `lat_syscall` with flag `null` (resp. `read` and `write`). Page fault corresponds to the LMBench `lat_pagefault`. The Shadow Page Fault is not from LMBench, it computes the time to solve a genuine page fault, i.e. the following sequence: after a first page fault, the hypervisor injects the fault to the guest, which adds a mapping in its GPT, then the execution faults again, and the hypervisor adds a mapping in the SPT.

In Table 6.2, we compare the overhead incurred by SecT hypervisor to the overheads incurred by Prosper and L4Linux presented in [Nem+15]. Note that the benchmarks

Benchmark	Native	Ours	Overhead
Simple Syscall	0.2568	2.9847	1062%
Simple Read	0.6722	4.4269	559%
Simple Write	0.6008	4.031	571%
Page Fault	1.5259	8.7215	472%
Shadow Page Fault	35.038	177.24	406%

TABLE 6.1: Low-Level Benchmarks

of SecT were performed on a Linux v4.2 whereas those of Prosper and L4Linux were performed on a Linux v2.6.34.

As can be seen, the hypervisor we study have better results than L4Linux, but Prosper is three times faster. It can be explained by two facts. Firstly, our hypervisor is still in the process of being optimized. Secondly, the use of SPT makes every access to memory slower, because a page fault from the guest triggers two page faults. SPTs are indeed not the fastest solution. Recall from Section 1.1.3 that the three ways of virtualizing memory are:

- Direct paging.
- Hardware virtualization extensions.
- SPT.

Direct paging and SPTs are the two *software* solutions. Direct paging implies more modification to the guest than SPTs, because the guest must cooperate with the hypervisor to modify its PTs. The SPTs solution is therefore more portable.

The *hardware* solution is the fastest solution. However virtualization extension are not available on all the platforms. For example, the cortex A9, which is one of ARM's most widely deployed and mature applications processors, does not provide virtualization extensions. Furthermore, the hardware is not proved. The behavior of the processor is largely documented, but the implementation is not open, and we have no guarantees that the behavior is indeed conform to the specifications. From a security point of view, the less we can rely on the unproved hardware, the better.

Benchmark	Ours	Prosper	L4Linux
Simple Syscall	1062%	342%	2955%
Simple Read	559%	155%	836%
Simple Write	571%	181%	874%

TABLE 6.2: Overhead Comparison

Synthetic benchmarks measure raw hardware performance. This kind of benchmarks was not performed on this hypervisor, because hypervisors do not have much impact on it [Nor+15].

6.2 Proofs

We measure the length and the difficulty of the proofs with *hints*. A hint corresponds to an interaction with the prover. After each interaction with the prover, the prover runs a

decision procedure to try to finish the proof. A hint is similar to applying a *tactic* followed by *auto* in Coq.

For example, *unfolding* a definition, *composing* with a lemma, *applying* a 'for all' statement to a particular element are hints. However, writing a comment in the proof and cleaning the goal (i.e. removing duplicated information) are also hints. They indeed correspond to interactions with the prover. Therefore, similarly to counting the number of line of code in some program or the number of lines of proofs in some proof assistants, counting hints is not a perfect metric but it gives a good idea of the difficulty of the proof. Especially for comparing one proof to another.

6.2.1 Example: Proof of Unmap Commutation

To give a more precise idea of the meaning of hints, we consider the proof of Lemma 5.5.3, and compare its pen-and-paper version to the mechanized version. Figure 6.1 show the beginning of the mechanized proof, where one case is hidden, and Figure 6.2 show the rest of the proof, from this case. We have written below a sketch of the pen-and-paper proof of Lemma 5.5.3, where we just show the cases and list the arguments that we have used. For each argument, we have indicated in square brackets the corresponding step in the mechanized proof.

We notice a first difference between the mechanized and pen-and-paper version, as the 6 first steps of Figure 6.1 do not appear in the pen-and-paper proof. This is only because in the mechanized version, we have stated the lemma using some wrappers, that we have to unfold. Then the case `a overflow` was not described in the pen-and-paper proof because we have not presented the reasoning about arithmetic, to simplify the presentation. The case which is showed in Figure 6.1 corresponds almost perfectly to the pen-and-paper case $pa \notin perm.priv(i)$.

The other case, showed in Figure 6.2, corresponds the pen-and-paper case $pa \in perm.priv(i)$. For readability, we have not presented the case $e \notin seg'(pa).tags \wedge e \in seg_1(pa).tags$ ($\texttt{e} \notin \texttt{tags}^0 \wedge \texttt{e} \in \texttt{ytags}$).

As could be expected, the mechanized proof presents more steps than the mathematical proof, for several reasons. Firstly, in the mechanized proof we need to destruct explicitly a structure to link it to its fields. Also, we have more wrappers around view functions in the mechanized version, in order to factorize code and proofs. These wrappers then need to be unfolded. Finally, given the way we have written our lemmas, the mechanized proof sometimes lacks factorization. For example, in the pen-and-paper proof, we have a case $v_e \notin [va_{pg}, va_{pg} + page\_size[$, in which we deduce that v_e is mapped in the new memory iff it is mapped in the old memory. It corresponds to two cases in the mechanized proof (`notin:`), one case where v_e is mapped in the two memories, one case where it is not mapped in the two memories.

Case $pa \notin perm.priv(i)$. [$\neg$`pa_in_priv_region(vcpu_idx, pa)`]
Definition 5.2.4 of private segment abstraction. [`Compose priv_segment_view_spec`]
Definition 5.5.4 of unmap private segment. [`Compose unmap_page_priv_proj`]

Case $pa \in perm.priv(i)$. [`pa_in_priv_region(vcpu_idx, pa)`]
Definition 5.2.4 of private segment abstraction. [`Compose priv_segment_view_spec`]
Definition 5.5.4 of unmap private segment. [`Compose unmap_page_priv_proj`]
Lemma 4.3.4, stating that memory byte values are unchanged [$\texttt{b1}^0 = \texttt{b1}$]
Deduce from $\mathcal{H}$ that tags are not equals [$\texttt{ytags} \neq \texttt{tags}^0$]
Specification of two sets are not equals [`Compose inequal_sets`]

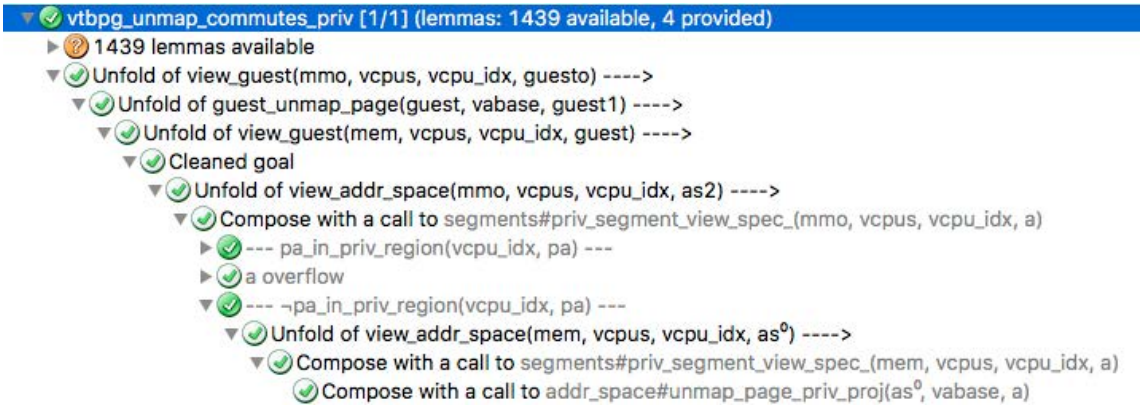


FIGURE 6.1: Commutation of Unmap - Case $pa \notin perm.priv$

Case $e \in seg'(pa).tags \wedge e \notin seg_1(pa).tags$. [e ∈ tags⁰ ∧ e ∉ ytags]

Case $v_e \in [va_{pg}, va_{pg} + page\_size[$. [in : [va, va + $size_{page}$[]
Lemma 4.3.3 about unmap effects [Compose vtlbpg_unmap_after_rw]
[Compose vtlbpg_unmap_after_ro]

Case $v_e \notin [va_{pg}, va_{pg} + page\_size[$. [in : [va, va + $size_{page}$[]
Lemma 4.3.3 about unmap effects [Compose vtlbpg_unmap_after_rw]
[Compose vtlbpg_unmap_after_ro]
Lemma 5.2.1 about tags specification [Applying forall view_tag_spec]
Definition 5.5.1 of remove tags [Compose remove_vas_tags_spec]
[Applying forall page_not_in_tags]

Case $e \notin seg'(pa).tags \wedge e \in seg_1(pa).tags$. [e ∉ tags⁰ ∧ e ∈ ytags]
...

6.2.2 Quantification of the Proof Effort

We list the hints for the main properties of the system.

Preservation of Invariants Table 6.3 show hints corresponding to the preservation of the invariants described in Chapter 4. For each operation, we also present the hints needed for proving the effects of the operation (Section 4.3). The proof of preservation of *unmap* and of its effects is short compared to the *map* and *unmap_all* counterparts. It can be explained because the *unmap* operation does not modify the SPT structure, i.e. it does not allocate or deallocate second level PT.

Commutation Table 6.4 references the hints of the commutation proofs. In Section 5.5, we have stated and showed the commutation for the guest transition and memory operations, for the private segment of one guest. These hints correspond to the commutation of the *whole abstract state*, i.e. the commutation over each operation for the private, shared segments and registers of each guest.

We have not described the commutation of the *inject_abt* operation and the page fault wrapper in this document, because it corresponds to fastidious but not very interesting proofs. Basically, when a page fault occurs in the guest, it corresponds either to an

FIGURE 6.2: Commutation of Unmap - Case $pa \in perm.priv$

Property	Hints
Preservation over map	3150
Map effets	1400
Preservation over *GuestTrans*	180
GuestTrans effets	260
Preservation over unmap	260
Unmap effets	200
Preservation over unmap all	2300
Unmap all effets	930

TABLE 6.3: Preservation of Invariants

Property	Hints
Commutation of map	1500
Commutation of *inject_abt*	300
Page Fault wrapper	600
Commutation of *GuestTrans*	550
Definition of *GuestTrans* (Section 5.3.2)	1300
Commutation of unmap	600
Commutation of unmap all	320

TABLE 6.4: Commutation Proofs

authentic page fault (*InjectFault* transition) or a shadow page fault (*PageFaultWithMMU* transition). The wrapper extends the proof of commutation of the operation to the whole transition.

Security Properties The hints for security properties are showed in Table 6.5. As can be seen, these proofs are small. This is due to the methodology used. The abstract state is indeed very close to the specifications. Therefore we prove isolation on a system which provides isolation by design, which is straightforward. The difficult part of the proof resides in the preservation and the commutation proofs.

Miscellaneous We have also proved various properties such as:

- Properties on PTs and translation of the virtual space, described in Section 4.1, which make 1650 hints.
- Properties on structures of the abstract level, 2100 hints.
- Properties about abstraction functions, 2100 hints (not counting the abstract guest transition definition).

Property	Hints
Integrity	250
Confidentiality	400

TABLE 6.5: Security properties

6.2.3 Hints to Time Spent on Proofs

The number of hints does not provides a precise idea of the time spent on a proof. Some proofs might be short but difficult to handle, because they require a good knowledge of the base of lemmas already proved. Other proofs are just a long and fastidious sequence of unfolds. We try to factorize hints, by writing sub lemmas, that can be used in several proofs. This process tend to reduce the number of hints, and the difficulty of a proof, but requires time, to think about the architecture and write the sub lemmas needed.

6.3 Proof Maintenance

Our proof is composed of several stages:

1. The proof of preservation of invariants over transitions of the concrete model (Sections 4.1 and 4.2).
2. The proof of the specifications of the effects of the transitions (Section 4.3).
3. The proof of commutation between concrete and abstract transitions (Section 5.5).
4. The proof of security properties on the abstract model (Section 5.4).

Stage 2 depends on stage 1, stage 3 depends on stage 2, and stage 4 depends on the abstract system. Therefore, small modifications on the concrete state are absorbed by the upper stages. For example, a modification in the concrete state which does not modify the effects of a transition will not affect the stage 3. The proof by abstraction method helps rendering the proof more robust.

For the same reason, using sub lemmas helps limiting the impact on the proofs of a change on the model.

Furthermore, we made extensive use of a feature in Prove & Run prover which helps adapting a broken proof. If the lemma is modified (add a condition, change the arity of a predicate, change the definition of a predicate etc...), the tool will try to adapt the old proof to the new lemma. It is possible to interact with the tool in order to modify or add some steps during the adaptation, to help the tool to find the proof. This feature was an immense help for the maintenance of the proof.

6.4 Conclusion

The performances measured by the benchmarks of the hypervisor show that the object of our study is not a toy hypervisor. It allows to run several OSes with good performances.

Our proof effort is of about 16000 hints, i.e. interactions with the prover. We have given the number of hints for some proofs that we have described in mathematical formalism in this manuscript, so that the reader can have an idea of the meaning of a hint. Furthermore, the modeling of the two levels presented in Chapters 3 and 5, and the design of the proof architecture, that we have presented in Chapters 4 and 5, represents an important part of the proof effort that we cannot measure with hints.

Conclusion

Formal proof allows a software developer to ensure with a high level of trust that a program meets its specifications. In this thesis, we have formally proved that the memory management of a hypervisor ensures isolation between guests memory and registers. It represents the first proof of isolation of a Shadow Page Table (SPT) algorithm. Furthermore, the methodology that we have used enhance the confidence in the formally proved properties and is more resilient to changes.

6.5 Summary

Our property of isolation was proved by abstraction. The particularity of our approach is that there is a huge step between the abstract and concrete models. We argued in Section 2.5 that this technique allows us to increase both the confidence in the meaning of our properties and in the fact that our model corresponds to the implementation.

The PTs are maintained by the hypervisor to manage access to memory. As we have explained in Section 4.1.2, the management of the PT is error-prone. Furthermore, it complicates any operation which tries to access or modify memory, because it requires to translate addresses before accessing their content. In our abstract model, described in Chapter 5, we do not have a notion of PT anymore. It makes the model more readable and the properties easier to express.

On the other hand, our concrete model, defined in Chapter 3, is close to the implementation. We clearly define our hypotheses on the hardware, and the transitions of the system. Contrary to the abstract model, the concrete model is highly dependent on the hypervisor considered, but we believe that our precise description would allow one to reuse this model as a basis for similar hypervisors.

We have discussed in Section 5.6 the impact of modifications on the concrete model on the design of the abstract model. We claim that our method makes our proofs more resilient to changes because as long as a change in the concrete model does not have impact on the effects of a transition (lemmas presented in Section 4.3), the impacts in the commutation proofs are minor (see proofs of commutation in Section 5.5).

A central part of our work is presented in Chapter 4. In this chapter, we define all the invariants of the concrete model, and how they depend on each other. The proof of the effects of the guest transition and memory operations, and therefore the commutation proofs, rely on these invariants.

6.6 Contributions

We have compared our methodology to those used in the Prosper and seL4 projects in Section 2.4.3. We recall briefly how we compare to these two projects.

In the Prosper hypervisor, Nemati et al. use the direct paging mechanism, not SPTs [Nem+15]. In the direct paging mechanism, the guest manages its own GPT. The hypervisor checks that the GPT verifies some properties before allowing the MMU to use it.

Therefore, all the invariants we have about the well-formedness of SPTs are not invariants of the GPTs in their model, but properties of the GPTs that the hypervisor checks before using them. Their abstraction stays close to the concrete model, they abstract the PTs of the hypervisor but keep the PT of the guest.

The seL4 kernel has not much in common with the hypervisor that we consider. First seL4 is not a hypervisor, so it does not manage SPTs. Furthermore, as we underlined in Section 2.2.1, the access right management is based on capabilities, whereas ours is not. We build our abstract model differently. In seL4, the notion of memory disappears in their highest levels of abstraction, their description is generic and thus very abstract. Our abstract model is merely a high level description of our concrete model, we have several abstract machines, in which the concept of registers and memory is still present, but highly simplified.

The particularity of our approach is that we have a concrete model very close to the C code, in which we uses low-level data structures, and an abstract system in which all the complex data structures have been abstracted.

The first advantage of such a method, as we have underlined in Section 2.5, is that it allows us to have a high level of assurance that the system enforces the stated properties. Indeed, the simple design of the abstract model allows us to state simple, easy to understand properties. Therefore we have few chances to state a property that does not have the intended meaning. Furthermore, our concrete model is close to implementation, it means that we can be confident that our concrete model corresponds to the implementation. It also means that we broaden the attack surface, for example, we have represented PTs as low-level structures located in memory, thus considering all the possible isolation breaches dues to a bad management of their structure in memory.

The second advantage of such a method is the robustness of the proof. As we have explained in Section 6.3, our proof architecture consists of four layers: the preservation of invariants on the concrete system, the specification of the effects of the concrete transitions, the refinement and the proof of security properties on the abstract state. Some changes in one layer can be confined to the next layer, without affecting the other upper layers. In particular, it means that changes in the concrete level algorithms which do not change the observable effects of a transition do not impact the proof of commutation and the proofs on the abstract model. This advantage is important, because the lack of robustness of proof is an obstacle to the use of formal methods for large systems in industry, as it makes the maintenance of the system costly and difficult.

Finally, in this manuscript we review in detail our models and methodology. We believe that this could be of great help for someone trying to start a proof of such a system. Among others, we provide a precise model of the ARMv7 processor, we present our invariants and show how they interact with each other. We show how we build our abstraction on top of a concrete state which verify all the invariants. In particular, we try to give the intuition about how to build such an abstract model. All in all, we explained all the architecture of the proof. During the development, we have first proved the commutation for the map operation, then we have applied the same method to the $unmap$, $unmap\_all$ and $inject\_abt$ operations and to the guest transition. We conclude that the proofs were easier to conduct once we had the architecture of the proof in mind.

6.7 Perspectives

Proofs From this work, several directions are possible. First of all, we could continue this work by verifying that the invariants hold after the initialization phase. Mainly,

after initialization, we need to verify that the HPTs are located in the hypervisor space, and that they map all the user and pool regions. We also need to verify that the SPTs are empty except for the forbidden range, which maps the exception handler and SPT regions correctly.

The virtualization of devices is done with PTs, just as memory virtualization is. We have not included the devices in our model. Extending the model would represent a small extension, as it uses the PTs mechanism already in place.

DMA allows a device to access the memory directly, bypassing the CPU and therefore the PTs. A device using DMA can therefore access any part of the memory and break isolation. The I/O MMU component (or SMMU) allows the hypervisor to control the access of devices to memory through PTs. With such extensions, it is possible to extend our model to DMA-aware devices, and to reuse most of our work on PTs.

Access to devices is linked to the management of IRQs, because devices may trigger hardware interrupt, which must be virtualized by the hypervisor for the guests. Throughout our modeling work, we realized that the management of IRQs, more precisely the virtualization of IRQs, was also a complex part of the hypervisor. Therefore, verifying the IRQ management part of the hypervisor would be an interesting and relevant future direction.

Concurrency The hypervisor we consider in this work runs on a uni-processor. Our proofs cannot be directly transposed to multiprocessor systems because we work with a sequential model of execution which is not valid anymore in the context of concurrency. Indeed, when several tasks can be executed simultaneously, there exist many possibilities for the atomic operations to be interleaved. The way the operations are interleaved may lead to different execution scenarios. When it comes to reasoning, one must be able to capture in the model all the possible execution scenarios. As more and more systems run on multiprocessors, studying how our model could be adapted to reason about concurrency would be a good way forward.

Application to a larger scope of systems One of the issue of formal proof is its lack of robustness against software modifications. As we have explained, the proof by abstraction helps solving this issue because the changes are absorbed by the several layers of proof. We could also combine this method with the use of tools which make the modification process easier. For example, the analysis presented in [AJL16] makes it possible to precisely delimit the fields of a structure which are modified by a transition. With such a tool, the impact of code modification on proofs can be reduced, thus making it possible to consider having a more mutable base of code.

Even if the methodology is robust regarding modifications, it is not good enough to consider making a full proof of a very large system, such as a general purpose OS, that change continuously. This methodology is preferred for systems which are unlikely to be modified very often. For a large system likely to be modified a lot, it is more realistic to consider isolating some stable critical parts of the system, and proving them. Indeed, when we change something in the concrete model, we must show that it is still conform to the specifications. This step can be costly if it has to be performed repeatedly. A solution to make proof by abstraction an acceptable solution for very large systems is to lower the cost of the link between the abstract and the concrete models. It can be done, for example, by establishing only a *semi* formal link between the two models (e.g. a pen and paper proof), or even by having just a *semi* formal concrete model (e.g. a diagram in a

defined semantic). The trust in the security properties would be significantly lowered, but combined with full proof of some critical parts, it could be an interesting option.

Glossary

CC Common Criteria. 14

CPSR Current Program Status Register. 40–42, 44

DMA Direct Memory Access. 39, 125

GIC Generic Interrupt Controller. 42, 47

GPT Guest Page Table. 7, 8, 43, 51, 52, 54, 65, 66, 112, 115

GVA Guest Virtual Address. 7, 8, 43, 45

HPT Hypervisor Page Table. 7, 38, 60–62, 65, 66, 73, 76, 125

HVA Hypervisor Virtual Address. 7, 8

IPA Intermediate Physical Address. 7, 8, 45, 50

MMU Memory Management Unit. 6–8, 21, 36, 41, 42, 45, 50, 64, 81

OS Operating System. 3–9, 11, 17–19, 21

PA Physical Address. 7, 8

PL0 Privileged Level 0. 40

PL1 Privileged Level 1. 40

PT Page Table. 6–8, 12, 13, 20, 21, 25, 29, 31, 33–37, 39, 41, 45, 46, 51, 52, 57, 59, 64, 123

RO Read Only. 37, 38

RW Read/Write. 37, 38, 86

SPSR Application Program Status Register. 44

SPT Shadow Page Table. 8, 20–22, 28, 29, 42, 43, 45, 50–52, 57, 59, 61, 62, 66, 69, 71, 73–75, 78, 80, 84, 88, 98, 112, 115, 125

TCB Trusted Computing Base. 4

TLB Translation Lookaside Buffer. 8, 42, 51

Bibliography

[Acs] *ANSI/ISO C Specification Language*. https://frama-c.com/acsl.html.

[ADAD14] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 0.80. Arpaci-Dusseau Books, 2014.

[Afl] *American Fuzzy Loop*. http://lcamtuf.coredump.cx/afl/.

[AJL16] Oana Fabiana Andreescu, Thomas Jensen, and Stéphane Lescuyer. "Correlating Structured Inputs and Outputs in Functional Specifications". In: *Software Engineering and Formal Methods: 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*. Ed. by Rocco De Nicola and Eva Kühn. Cham: Springer International Publishing, 2016, pp. 85–103. ISBN: 978-3-319-41591-8. DOI: 10.1007/978-3-319-41591-8_7. URL: http://dx.doi.org/10.1007/978-3-319-41591-8_7.

[Alk+10] Eyad Alkassar et al. "Automated Verification of a Small Hypervisor". In: *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*. Vol. 6217. Lecture Notes in Computer Science. Edinburgh, UK: Springer, Aug. 2010, pp. 40–54.

[Alk+12] E. Alkassar et al. "Verification of TLB Virtualization Implemented in C". In: *4th International Conference on Verifed Software: Theories, Tools, and Experiments, VSTTE'12*. Lecture Notes in Computer Science. Philadelphia, USA: Springer-Verlag, 2012. URL: http://www-wjp.cs.uni-saarland.de/publikationen/ACKP12.pdf.

[Alm+11] José Bacelar Almeida et al. *Rigorous Software Development - An Introduction to Program Verification*. Undergraduate Topics in Computer Science. Springer, 2011, pp. I–XII, 1–263. ISBN: 978-0-85729-018-2.

[Alt] *Alt-Ergo by OCamlPro*. https://alt-ergo.ocamlpro.com/.

[And13] Sarah J. Andrabi. *Verification of XMHF HPT Protection Setup*. Tech. rep. University of North Carolina, 2013. URL: http://cs.unc.edu/~sandrabi/Project_work/VerificationofXMHFHPTProtectionSetup.pdf.

[Arm] *ARM Processor Architecture*. 2016. URL: http://www.arm.com/products/processors/instruction-set-architectures/.

[Bar+12] G. Barthe et al. "Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization". In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. 2012, pp. 186–197.

[Bau+11] Christoph Baumann et al. "Proving Memory Separation in a Microkernel by Code Level Verification". In: *1st International Workshop on Architectures and Applications for Mixed-Criticality Systems (AMICS 2011)*. Ed. by Wilfried Steiner and Roman Obermaisser. To appear. Newport Beach, CA, USA: IEEE Computer Society, Mar. 2011. URL: http://www-wjp.cs.uni-saarland.de/publikationen/Baumann-AMICS2011.pdf.

[BB09] Christoph Baumann and Thorsten Bormer. "Verifying the PikeOS Microkernel: First Results in the Verisoft XT Avionics Project". In: *Doctoral Symposium on Systems Software Verification (DS SSV 2009)*. Ed. by Ralf Huuck, Gerwin Klein, and Bastian Schlich. Aachener Informatik Berichte AIB-2009-14. Department of Computer Science, RWTH Aachen, June 2009, pp. 20–22. URL: http://aib.informatik.rwth-aachen.de/2009/2009-14.pdf.

[Bev89] W. R. Bevier. "Kit: A Study in Operating System Verification". In: *IEEE Trans. Softw. Eng.* 15.11 (1989), pp. 1382–1396. ISSN: 0098-5589. DOI: 10.1109/32.41331. URL: http://dx.doi.org/10.1109/32.41331.

[BJS16] Pauline Bolignano, Thomas Jensen, and Vincent Siles. "Modeling and Abstraction of Memory Management in a Hypervisor". In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 214–230. DOI: 10.1007/978-3-662-49665-7_13. URL: http://dx.doi.org/10.1007/978-3-662-49665-7_13.

[Bla+15] Allan Blanchard et al. "A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C". In: *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*. 2015, pp. 15–30. URL: http://dx.doi.org/10.1007/978-3-319-19458-5_2.

[Bru+11] David Brumley et al. "BAP: A Binary Analysis Platform". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Snowbird, UT: Springer-Verlag, 2011, pp. 463–469. ISBN: 978-3-642-22109-5. URL: http://dl.acm.org/citation.cfm?id=2032305.2032342.

[Cbm] *Bounded Model Checker for C and C++ programs*. http://www.cprover.org/cbmc/.

[Cc] *Common Criteria Portal*. http://www.commoncriteriaportal.org/.

[Chi07] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. First. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780132349710.

[Coh+09] Ernie Cohen et al. "VCC: A Practical System for Verifying Concurrent C". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*. Vol. 5674. Lecture Notes in Computer Science. Springer, 2009, pp. 23–42. ISBN: 978-3-642-03358-2. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=117859.

[Coq] *The Coq proof assistant reference manual, version 8.2*. August 2009.

[Dam+13] Mads Dam et al. "Formal verification of information flow security for a simple arm-based separation kernel". In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. 2013, pp. 223–234. DOI: 10.1145/2508859.2516702. URL: http://doi.acm.org/10.1145/2508859.2516702.

[DBK14a] Matthias Daum, Nelson Billing, and Gerwin Klein. "Concerned with the unprivileged: user programs in kernel refinement". In: *Formal Aspects of Computing* 26.6 (2014), pp. 1205–1229. ISSN: 1433-299X. DOI: 10.1007/s00165-014-0296-9. URL: http://dx.doi.org/10.1007/s00165-014-0296-9.

[DBK14b] Matthias Daum, Nelson Billing, and Gerwin Klein. "Concerned with the unprivileged: user programs in kernel refinement". In: *Formal Asp. Comput.* 26.6 (2014), pp. 1205–1229. URL: http://dx.doi.org/10.1007/s00165-014-0296-9.

[DGN13] Mads Dam, Roberto Guanciale, and Hamed Nemati. "Machine Code Verification of a Tiny ARM Hypervisor". In: *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*. TrustED '13. Berlin, Germany: ACM, 2013, pp. 3–12. ISBN: 978-1-4503-2486-1. DOI: 10.1145/2517300.2517302. URL: http://doi.acm.org/10.1145/2517300.2517302.

[DN10] Christoffer Dall and Jason Nieh. "KVM for ARM". In: *2010 Ottawa Linux Symposium, July 2010*. 2010, pp. 45–56. URL: http://www.cs.columbia.edu/~cdall/pubs/ols2010-paper.pdf.

[Do1] *SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION*. URL: http://sesam.smart-lab.se/IG_Prgsak/Publikat/ED12B_DO178B.pdf.

[Dut14] Bruno Dutertre. "Yices 2.2". In: *Computer-Aided Verification (CAV'2014)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 737–744.

[Esx] *VMware ESXi*. 2016. URL: https://www.vmware.com/fr/products/esxi-and-esx/overview.

[Eve] *Home of Event-B and the Rodin Platform*. http://www.event-b.org/contact.html.

[FN79] Richard J. Feiertag and Peter G. Neumann. "The foundations of a provably secure operating system (PSOS)". In: *IN PROCEEDINGS OF THE NATIONAL COMPUTER CONFERENCE*. AFIPS Press, 1979, pp. 329–334.

[HT05] M. Hohmuth and H. Tews. "The VFiasco approach for a verified operating system". In: *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*. 2005.

[Iom] *AMD I/O Virtualization Technology (IOMMU) Specification*. 2015. URL: http://support.amd.com/TechDocs/48882_IOMMU.pdf.

[Isa] *Isabelle*. https://isabelle.in.tum.de/.

[Jac12] Stephen Jacklin. *Certification of Safety-Critical Software Under DO-178C and DO-278A*. 2012. URL: https://ti.arc.nasa.gov/publications/5357/download/.

[Kle+09] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[Kle09] Gerwin Klein. "Operating System Verification — An Overview". In: *Sādhanā* 34.1 (2009), pp. 27–69.

[Kov13] Mikhail Kovalev. "TLB virtualization in the context of hypervisor verification". eng. PhD thesis. Postfach 151141, 66041 Saarbrücken: Universität des Saarlandes, 2013. URL: http://scidok.sulb.uni-saarland.de/volltexte/2013/5215.

[KSD13] Narges Khakpour, Oliver Schwarz, and Mads Dam. "Machine Assisted Proof of ARMv7 Instruction Level Isolation Properties". In: *Certified Programs and Proofs: Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Ed. by Georges Gonthier and Michael Norrish. Cham: Springer International Publishing, 2013, pp. 276–291. ISBN: 978-3-319-03545-1. DOI: 10.1007/978-3-319-03545-1_18. URL: http://dx.doi.org/10.1007/978-3-319-03545-1_18.

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370. ISBN: 3-642-17510-4, 978-3-642-17510-7. URL: http://dl.acm.org/citation.cfm?id=1939141.1939161.

[Ler09] Xavier Leroy. "A formally verified compiler back-end". In: *Journal of Automated Reasoning* 43.4 (Dec. 2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4. URL: https://hal.inria.fr/inria-00360768.

[Les15] Stéphane Lescuyer. "ProvenCore: Towards a Verified Isolation Micro-Kernel". In: *International Workshop on MILS: Architecture and Assurance for Secure Systems*. 2015. URL: http://mils-workshop-2015.euromils.eu/downloads/hipeac_literature/04-mils15_submission_6.pdf.

[Lie95] J. Liedtke. "On Micro-kernel Construction". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 237–250. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224075. URL: http://doi.acm.org/10.1145/224056.224075.

[Lmb] *LMbench*. http://www.bitmover.com/lmbench/.

[LS09] Dirk Leinenbach and Thomas Santen. "Verifying the Microsoft Hyper-V Hypervisor with VCC". In: *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. Ed. by Ana Cavalcanti and Dennis R. Dams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 806–809. ISBN: 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_51. URL: http://dx.doi.org/10.1007/978-3-642-05089-3_51.

[LV95] N. Lynch and F. Vaandrager. "Forward and Backward Simulations". In: *Information and Computation* 121.2 (1995), pp. 214 –233. ISSN: 0890-5401. DOI: http://dx.doi.org/10.1006/inco.1995.1134. URL: http://www.sciencedirect.com/science/article/pii/S0890540185711340.

[MN11] Roberto Mijat and Andy Nightingale. *Virtualization is Coming to a Platform Near You*. 2011. URL: https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf.

[Mos+09] Michal Moskal et al. *A Practical Verification Methodology for Concurrent Programs*. Tech. rep. 2009. URL: https://www.microsoft.com/en-us/research/publication/a-practical-verification-methodology-for-concurrent-programs/.

[Mur+12] Toby C. Murray et al. “Noninterference for Operating System Kernels”. In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*. 2012, pp. 126–142. DOI: 10.1007/978-3-642-35308-6_12. URL: http://dx.doi.org/10.1007/978-3-642-35308-6_12.

[Mur+13] Toby C. Murray et al. “seL4: From General Purpose to a Proof of Information Flow Enforcement”. In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 2013, pp. 415–429. URL: http://dx.doi.org/10.1109/SP.2013.35.

[Nem+15] Hamed Nemati et al. “Trustworthy Memory Isolation of Linux on Embedded Devices”. In: *Trust and Trustworthy Computing - 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings*. 2015, pp. 125–142. DOI: 10.1007/978-3-319-22846-4_8. URL: http://dx.doi.org/10.1007/978-3-319-22846-4_8.

[NF03] P. G. Neumann and R. J. Feiertag. “PSOS revisited”. In: *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*. 2003, pp. 208–216. DOI: 10.1109/CSAC.2003.1254326.

[NGD15] Hamed Nemati, Roberto Guanciale, and Mads Dam. “Trustworthy Virtualization of the ARMv7 Memory Subsystem”. In: *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*. 2015, pp. 578–589. URL: http://dx.doi.org/10.1007/978-3-662-46078-8_48.

[Nor+15] Jan C. Nordholz et al. “XNPro: Low-Impact Hypervisor-Based Execution Prevention on ARM”. In: *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices, TrustED 2015, Denver, Colorado, USA, October 16, 2015*. 2015, pp. 55–64. DOI: 10.1145/2808414.2808415. URL: http://doi.acm.org/10.1145/2808414.2808415.

[Pea] *Peach Fuzzer*. http://www.peachfuzzer.com/.

[PG74] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures.” In: *Commun. ACM* 17.7 (1974), pp. 412–421. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm17.html#PopekG74.

[Pop+79] Gerald J. Popek et al. “UCLA Secure UNIX”. In: *Managing Requirements Knowledge, International Workshop on* 0 (1979), p. 355. DOI: http://doi.ieeecomputersociety.org/10.1109/AFIPS.1979.128.

[Qem] *QEMU, Open Source Processor Emulator*. 2016. URL: http://wiki.qemu.org/Main_Page.

[Ric10] Raymond J. Richards. “Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel”. In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by S. David Hardin. Boston, MA: Springer US, 2010, pp. 301–322. ISBN: 978-1-4419-1539-9. DOI: 10.1007/978-1-4419-1539-9_10. URL: http://dx.doi.org/10.1007/978-1-4419-1539-9_10.

[Rus92] John Rushby. *Noninterference, Transitivity, and Channel-Control Security Policies*. Tech. rep. 1992. URL: http://www.csl.sri.com/papers/csl-92-2/.

[Sec] *Security in Telecommunication, Technische Universität Berlin*. http://www.isti.tu-berlin.de/security_in_telecommunications/.

[Sew+11] Thomas Sewell et al. "seL4 Enforces Integrity". In: *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*. 2011, pp. 325–340. DOI: 10.1007/978-3-642-22863-6_24. URL: http://dx.doi.org/10.1007/978-3-642-22863-6_24.

[SGG12] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 9th. Wiley Publishing, 2012. ISBN: 978-1-118-06333-0.

[SH02] Jonathan S. Shapiro and Norman Hardy. "EROS: A Principle-Driven Operating System from the Ground Up". In: *IEEE Software* 19.1 (2002), pp. 26–33. DOI: 10.1109/52.976938. URL: http://dx.doi.org/10.1109/52.976938.

[Smm] *ARM System Memory Management Unit*. 2012. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0062b/index.html.

[Vas+13] Amit Vasudevan et al. "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 430–444. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.36. URL: http://dx.doi.org/10.1109/SP.2013.36.

[Vbx] *Virtual Box*. 2016. URL: https://www.virtualbox.org/.

[Vet+15] Julian Vetter et al. "Uncloaking Rootkits on Mobile Devices with a Hypervisor-Based Detector". In: *Information Security and Cryptology - ICISC 2015 - 18th International Conference, Seoul, South Korea, November 25-27, 2015, Revised Selected Papers*. 2015, pp. 262–277. DOI: 10.1007/978-3-319-30840-1_17. URL: http://dx.doi.org/10.1007/978-3-319-30840-1_17.

[Vir] *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. 2007. URL: https://www.vmware.com/files/pdf/VMware_paravirtualization.pdf.

[Why] *Why3, Where Programs Meet Provers*. http://why3.lri.fr/.

[WKP80] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. "Specification and Verification of the UCLA Unix Security Kernel". In: *Commun. ACM* 23.2 (1980), pp. 118–131. DOI: 10.1145/358818.358825. URL: http://doi.acm.org/10.1145/358818.358825.

[Ws] *VMware Workstation Pro*. 2016. URL: https://www.vmware.com/fr/products/workstation.

[X86] *Intel® 64 and IA-32 Architectures Software Developer Manuals*. 2016. URL: http://www.arm.com/products/processors/instruction-set-architectures/.

[Xen] *Xen Project Software Overview*. 2016. URL: http://wiki.xen.org/wiki/Xen_Project_Software_Overview.

[Z3] *The Z3 Theorem Prover*. http://rise4fun.com/Z3/tutorial/guide.

[Zha+16] Yongwang Zhao et al. "Reasoning About Information Flow Security of Separation Kernels with Channel-Based Communication". In: *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 791–810. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_50. URL: http://dx.doi.org/10.1007/978-3-662-49674-9_50.