Stage Master Recherche

Rapport Bibliographique

La Sécurité des Unikernels

Domaines: Sécurité - Système d'Exploitation

Auteur:
Jean-Joseph Marty

Encadrants:
Jean-Pierre Talpin
Nikolaos Parlavantzas
Time, Events and Architectures

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Abstract

De nos jours, la conception des systèmes est en train de s'adapter pour pouvoir unifier deux mondes complètement opposés : le monde du *Cloud* avec ses réseaux de serveurs et le monde de l'embarqué avec ses objets connecté disposant de ressources très limitées. La mise en place de tels systèmes nécessite de nombreuses précautions à cause de la criticité de leur rôle. Nous aborderons quels moyens de sécurité disposent les concepteurs de ces systèmes, puis quelle partie du système doit être sécurisée et enfin nous parlerons de la notion de contrôle d'accès.

Table des matières

Introduction

La sécurité des systèmes informatiques est aujourd'hui devenue très complexe. En effet, la multiplication des architectures, des systèmes, des composants et des librairies tierces rendent l'exécution d'une application, vérifiée ou non, périlleuse et difficile. Les failles et vulnérabilités du système construit autour de l'application sont aussi nombreuses que les multiples inter-connections entre chacun des composants hétérogènes qui le constituent.

A cette préoccupation, le principe d'*unikernel* apparaît comme une solution qui simplifie aussi bien le développement d'un logiciel que son audit de qualité ou de sécurité. Bien qu'issu du monde de l'informatique embarqué, où il était entendu comme librairie système, la notion d'*unikernel* trouve son essors dans ses application au "*cloud computing*" : minimisation de la maintenance et des besoins énergétiques. Son rival, Docker, est bien plus simple à administrer mais souffre de problème de sécurité puisqu'il repose sur un système d'exploitation hôte. Cette différence permet à l'*unikernel* d'assurer un cloisonnement plus précis entre les applications (machines virtuelles), et par conséquent de limiter la contagion d'une attaque ou la propagation d'une panne.

Le rôle de l'*unikernel* est de fournir un ensemble de fonctionnalités basiques pour permettre la mise en place et l'exécution d'une application donnée. Ces fonctionnalité peuvent permettre l'accès à des opérations spécifiques à la plateforme ciblée dans le cadre de l'embarqué alors que dans le cadre du *Cloud* celles-ci sont standardisées (par le biais d'un hyperviseur matériel). Ainsi, à la différence d'un système d'exploitation complet, donc lourd et complexe, l'*unikernel* reste léger et simple, ce qui lui donne alors une probabilité a priori bien plus faible de posséder des failles de sécurité, de part sa superficie réduite. Dans cette bibliographie, nous allons parler à la fois de mécanismes de sûreté et de sécurité. En effet, ces deux notions sont complémentaires mais pas identiques.

Le sujet de stage se déroulera en trois parties. Tous d'abord, une analyse des mécanismes de sécurité actuels pour les concepteurs ou administrateurs d'*unikernel*. Ensuite, une partie sera dédiée à la recherche des différents moyens de positionner la zone de code de confiance (*Trusted Computing Base*) au moyen de méthodes de vérification et de preuve à même de passer à l'échelle (d'une librairie système complète). Enfin, une étude de cas visant à mettre en pratique une méthodologie de preuve par l'exploration d'une solution de système de gestion des droits d'accès simple sera abordée ainsi qu'une expérimentation avec une preuve de ce système seront réalisées à des fins éducatives.

Dans cette bibliographie, nous aborderons tout d'abord un état de l'art afin de préciser, l'avancement scientifique actuel du domaine. Puis nous détaillerons les thématiques de recherche qui constituent le sujet de stage. Ceci nous permettra de comprendre ce que le stage apporte à l'équipe encadrante mais aussi de voir quelles directions potentielles nous pouvons prendre pour apporter une contribution à la science. Enfin nous montrerons les applications potentielles de ces différentes thématiques dans l'état actuel du domaine.

Les Unikernels

Un *unikernel* se définit comme étant une nouvelle manière de voir l'informatique système. En effet, de la même manière que le paradigme programmation fonctionnelle ou objet qui apportent un regard nouveau dans la manière de représenter le code, un *unikernel* représente une nouvelle place du système d'exploitation dans un système. En effet, puisque le système est une simple librairie propre à la plateforme ciblée, un programme n'est plus vu comme étant un composant du système d'exploitation car c'est le système d'exploitation qui est vu comme un composant d'un programme [1].

Mécanismes de Sécurité

Bien qu'il existe des méthodes d'analyses de la sécurité (comme *EBIOS* [2]) et des façons de concevoir un programme de manière à intégrer la sécurité dans le développement, nous aborderont dans le cadre de ce stage uniquement les mécanismes automatiques propres aux systèmes d'exploitations ou à la finalisation d'un programme. Nous entendons par là deux catégories : la protection *apriori* par une certification du code c'est-à-dire avant l'exécution du programme, et la protection *in tempore* par la vérification du respect des prédicats de sécurité, c'est-à-dire pendant l'exécution du programme. Il est important de remarquer que la certification définit de manière formelle la validité des prédicats à l'exécution.

La protection *apriori*, consiste à s'assurer qu'un programme est sûr et sécurisé. Pour qu'il soit considéré comme sécurisé, nous devons avoir des prédicats qui assurent que le programme fait uniquement ce pour quoi il est prévu.

La protection *in tempore*, consiste à faire respecter des conditions de sécurité et ce, à n'importe quel moment pendant l'exécution. Il peut s'agir par exemple du confinement en mémoire ou encore de l'analyse des appels systèmes pour déterminer si un programme est malveillant.

Concept de la TCB

La TCB (*Trusted Computing Base*) ou zone de confiance, est un ensemble de composants matériels ou logiciels critiques dont le dysfonctionnement, même mineur, impacte le système tout entier [3]. Aussi, la TCB est généralement la partie qui sera protégée et certifiée à défaut de pouvoir certifier et sécuriser l'ensemble du système. Elle permet normalement de vérifier, un ou plusieurs prédicats comme le confinement des processus.

Système de contrôle d'accès

Dans le cadre du stage, un modèle de contrôle d'accès sera étudié afin d'apprendre la démarche scientifique de la recherche au prototype en passant par l'étude théorique. Le but de cet exercice est d'explorer un système de contrôle d'accès, de prouver que son fonctionnement respectera certaines propriétés et qu'il reste exploitable pour des architectures embarquées ou disposant de peu de ressources.

1 État de l'art

1.1 Les Unikernels

1.1.1 Présentation

LibOS Les librairies système d'exploitation sont une forme spéciale de système d'exploitation qui se présente et s'utilise comme une librairie normale. D'un côté, elles apportent la manipulation de la machine (c'est à dire dialoguer avec les composants matériels), et de l'autre, une interface de fonctionnalités abstraites [4, 5]. L'avantage d'une *libOS* est tout d'abord la simplicité face à un système standard. En effet, cette dernière ne contient que le strict minimum, ce qui permet de réduire la surface du code et donc de réduire également le risque de failles, tout en simplifiant les audits. Aussi, cette réduction allège le nombre d'appels de code pour exécuter une fonction.

Unikernel Les *unikernels* sont des *libOS* qui permettent d'avoir une certaine modularité dans la création d'applications autonomes. À l'heure actuelle, la majorité des *unikernels* sont principalement utilisés pour des serveurs du *Cloud*. Certains ont un fonctionnement "à la demande". C'est-à-dire qu'il y a une exécution de l'*unikernel* par requête sur ce dernier.

Un des gros désavantages des *unikernels* face à des systèmes de virtualisation comme *Docker* est que, à l'heure actuelle, le programme doit être complètement réécrit pour pouvoir être adapté à la *LibOS* que représente l'*unikernel* puisqu'il n'existe pas encore de standard pour les *LibOS*, ce qui oblige donc à bien choisir l'*unikernel* cible ou à réécrire le code car l'*unikernel* peut avoir des contraintes au niveau du code source. En effet, certains *unikernels* n'acceptent qu'un nombre très réduit de langages.

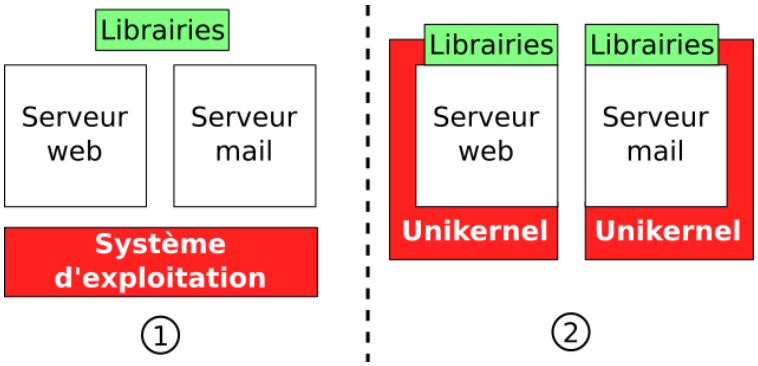

FIGURE 1 – Différence entre les systèmes : 1. système classique et 2. l'*unikernel*

Exokernel Il est important de ne pas confondre un *exokernel* avec un *unikernel*. Bien que très ressemblant, l'*exokernel* diffère puisqu'il sépare la notion de sécurité et l'abstraction matérielle. En effet, l'*exokernel* ne s'occupe que de la partie sécurité alors que l'abstraction matérielle est fournie par une librairie ou le programme lui-même. L'abstraction matérielle est un processus d'abstraction qui permet de ne plus être dépendant de l'architecture. Nous pouvons citer Nemesis qui est un *exokernel* développé par l'université de Cambridge [6, 7].

Microkernel Un microkernel est une version réduite d'un kernel standard afin d'avoir le nombre minimum de fonctionnalités nécessaires. Cela confère au *microkernel* une facilité de mise en place et de maintenance. Cette manière de concevoir les systèmes d'exploitation est utilisée pour concevoir les *hyperviseur*, que nous présenterons après [8]. Il est important de noter que la principale différence entre un *unikernel* et un *microkernel* est le nombre d'application. En effet, un *unikernel* gère une application unique à l'inverse du *microkernel* qui en gère un nombre variable généralement dynamique.

IoT Dans le cadre de l'Internet des Objets (*Internet of Things*), l'*unikernel* se positionne comme un moyen de déployer des applications complexes comme un serveur web de manière simplifiée puisque le développeur fait en partie abstraction de la partie interaction avec le matériel. Il est donc possible d'écrire des programmes embarqués avec du code de haut niveau de la même que OCaml (*MirageOS* permet de cibler des composants embarqués comme CUBIEBOARD 2). Cependant, le développement en est simplifié puisqu'il se réalise en faisant abstraction de la plateforme ciblée [9].

Certains *unikernels* comme *Riot* ou *Contiki* permettent de développer des applications sur des micro-contrôleurs (processeur réduit utilisé en embarqué). Ils permettent l'écriture de code en C et nécessitent très peu de mémoire (moins de 2Ko). Toutefois, il est intéressant de noter que *Contiki* est bien plus orienté réseaux *adhoc* que *Riot* puisqu'il facilite la création de programmes distribués avec le simulateur *Cooja* [10, 11].

HyperViseur Dans le cadre de l'informatique en nuage (*Cloud Computing*), l'*hyperviseur* est une machine destinée à distribuer des ressources entre les différents serveurs virtualisés. Ces derniers sont très populaires grâce à leur faible coût de mise en place et de maintenance. L'*hyperviseur* est un système qui virtualise des machines. Cependant, un des points sur lequel un *hyperviseur* peut représenter une menace dans la sécurité de ces serveurs est le cloisonnement entre les différentes machines virtuelles. En effet, si les machines ne sont pas complètement cloisonnée, une attaque peut être réalisée depuis une machine corrompue. Dans ce cas, la machine ciblée n'a que très peu de moyens car l'attaquant a accès aux informations stockées dans des couches logicielles (en mémoire par exemple) qui ne sont pas contrôlées par la victime [12].

Xen Cet *hyperviseur* est un *microkernel* qui fournit des fonctions basiques pour la gestion des machines virtuelles appelées dans ce contexte "domaine". Aussi, le "dom0" est la première machine virtuelle à être lancée et est également la seule pouvant accéder directement au matériel. Les autres instances de machines virtuelles sont lancées avec moins de privilèges. Dans la figure 2, le "dom0" est une instance qui possède un accès privilégié tandis que la machine virtuelle et les *unikernels* sont instanciés avec moins de privilèges [12]. Dans la figure 2, la partie "core" est le *microkernel* qui instancie "dom0" qui enfin instancie les machines du "domUser". Il y a dans le "domUser" une machine virtuelle et deux *unikernels*.

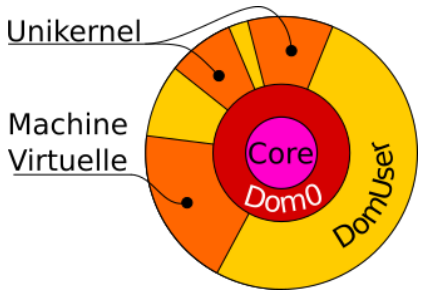

FIGURE 2 – Cartographie de Xen

1.1.2 Exemples d'Unikernels

MirageOS Cet *unikernel* utilise un langage de haut-niveau OCaml. L'utilisation de ce langage est intéressant pour plusieurs raisons :

- Langage multi-paradigmes qui exprime en peu de lignes des programmes complexes
- Le code compilé est très optimisé à l'exécution
- Langage bien typé et prouvable puisque la preuve de code OCaml est possible avec *Coq*

Aussi, il est possible de construire des applications pour Unix et Xen puisque cet *unikernel* cible actuellement le *Cloud*. *MirageOS* utilise bien moins de ressources qu'un serveur normal Linux qui

nécessite au moins 256Mo contre 32Mo pour un *unikernel MirageOS* [13].

ClickOS Cet *unikernel* cible principalement l'*hyperviseur* Xen, ce qui a pour conséquence de cibler uniquement le *Cloud*. Contrairement à *MirageOS*, *ClickOS* accepte les langages respectant l'interface *SWIG*. Cela permet d'écrire le code en Python ou en Perl par exemple, et de ne pas être forcé à utiliser qu'un seul langage. Une particularité propre à cet *unikernel* est d'améliorer la liaison entre les drivers du matériel avec la librairie ClickOS [14], ce qui a pour but d'accélérer le traitement des paquets réseaux.

Rump Dans le but de déboguer plus facilement le kernel de netBSD, une première idée selon laquelle il fallait développer des drivers dans l'espace utilisateur plutôt que dans l'espace kernel a émergée. Puis, afin de ne pas avoir de portage à réaliser, est venue la nouvelle idée de développer un drivers dans l'espace utilisateur puis de l'exécuter sans modification dans l'espace noyau. C'est pour cela que *Rump* apporte les fonctionnalités minimales au driver pour qu'il puisse fonctionner comme au niveau du noyau de netBSD. Plus tard, est apparu l'intérêt de glisser de cet outil de développement vers l'*unikernel*. En effet, il devint possible de créer des applications avec le minimum de code. Par exemple, un serveur dynamique qui serait embarqué soit sur un *hyperviseur* soit sur un appareil *IoT*. Contrairement à *ClickOS* et *MirageOS*, *Rump* nécessite uniquement une machine qui accepte du code compilé C99 et quelques centaines de Kilo octets de mémoire [15].

1.2 Mécanismes de sécurité

1.2.1 Sécurité par la preuve

A la manière de la preuve en logique mathématique, la preuve de programme permet de s'assurer que des prédicats soient bien respectés tout au long de l'exécution du programme. Le but est de minimiser les erreurs en ciblant les zones critiques (les *TCB*) afin de s'assurer que dans ces zones le code aura toujours le comportement souhaité. Dans le cas de logiciels très critiques c'est l'ensemble du code qui sera prouvé. Dans le cas de la preuve de programme nous nous intéressons à la démonstration assistée par ordinateur. Il existe différents outils qui permettent de faire de la preuve, cependant il n'ont pas tous la même manière d'aborder la démonstration.

Aussi, dans cette partie nous parlons de deux démarches : la vérification qui permet de vérifier qu'un programme fait ce qui est attendu de lui et la preuve qui prouve qu'un programme respecte certains invariants sur la base de prédicats. Comme la preuve est plus précise dans le cadre de la sécurité informatique, c'est pourquoi nous nous orientons plus vers ce mécanisme.

Il existe d'autres approches qui permettent de vérifier un programme : nous pouvons citer la BDD (*Binary Decision Diagram*) ou encore le problème SAT (satisfaisabilité booléenne). Ces méthodes peuvent dans certains cas prouver qu'un programme est correct mais de manière limitée. Bien que ces méthodes furent longtemps utilisée, nous ne traiterons que la vérification formelle qui semble ne pas avoir les même limites [16].

Coq L'assistant de preuve *Coq* utilise le langage de programmation fonctionnelle *Gallina*. Ce langage est basé sur le calcul des constructions (qui est une forme de lambda calcul typé). *Gallina* étend ce calcul en permettant des structures de données inductives. C'est cet assistant de preuve qui a prouvé le noyau de *CertikOS* ainsi que la *TCB* de *PipCore*.

F* Similaire au langage ML, ce langage est un langage fonctionnel dont le but est la vérification de programmes. Il supporte le polymorphisme, la dépendance des types, la monade (émuler des notions propres aux langages impératifs comme la notion d'état mémoire), le raffinement des types et le calcul de la plus faible précondition. La vérification de type dans F* s'appuie sur des solveurs SMT (une extension de SAT) et de preuve manuelle. Le principe de F* est de vérifier les préconditions et les postconditions d'une fonction. Pour y arriver F* tente de prouver le code en prouvant l'intégralité du code de la fonction. Il permet d'exporter le code vérifié vers du OCaml, F# ou C.

Haskell Créé en 1990, le langage de programmation fonctionnelle *Haskell* est un langage basé sur le lambda-calcul et la logique combinatoire. Ce langage évalue paresseusement les expressions, permet l'inférence de type ainsi que les listes en compréhension. Cette dernière fonctionalité permet de définir le contenu d'une liste par filtrage grâce à une règle ou grâce au contenu d'une autre règle (voir figure 3 : où depuis la liste des entiers est filtré uniquement les nombre inférieurs ou égaux à 42) [17]. Aussi, il existe un *unikernel* supportant ce langage : *HaLVM*. Cependant, nous nous intéressons plus à la partie preuve de *Haskell* [18].

$$\{n \in \mathbf{N} \mid n \leq 42\}$$

FIGURE 3 – Exemple de liste en compréhension [17]

Liquid Haskell est une extension de Haskell qui permet d'ajouter (automatiquement ou manuellement) des invariants logiques sur les types (il s'agit d'un raffinement) afin de déterminer si ils sont respectés ou pas. Cela donne une vision plus claire du code dangereux pendant le développement. *Liquid Haskell* aide le développement sur différents aspects comme vérifier la totalité des fonctions (détermine si une fonction gère bien tout les cas sur les entrées), vérifier le respect des intervalles dans les index de tableau, éviter les boucles infinies ou encore vérifier les prédicats pendant l'écriture du code... [19]

Langage B Ce langage propre à la méthode B est un langage pour vérifier des programmes. Le principe est de définir formellement le système et ses contraintes puis de se rapprocher d'un système technique par raffinement et ce de manière successive. Cela permet à la fin du processus de vérification de générer un code Ada ou C. C'est cette méthode de vérification de programme qui a servi à vérifier le programme de pilotage automatique (appelé METEOR) de la ligne 14 du métro de Paris.

1.2.2 Outils d'aide à la correction

Dans de nombreux cas, des erreurs surviennent lors de l'implémentation et ce indépendamment d'une preuve du code (dans le cas de la cryptographie il arrive qu'un défaut d'implémentation remette en cause la sécurité). En effet, il n'est pas rare de devoir mettre à jour un programme car l'API a évoluée ou parce qu'un bug a été découvert. Aussi, dans ce cas un développeur peut facilement passer à côté d'erreurs qui devrait être corrigées. Dans cet état de l'art nous présenterons deux outils, l'un permettant la compréhension du code à l'exécution, l'autre un outil d'aide à la modification de code.

Frama-C Cet outil permet le déboggage avancé du code. En effet, il ne se limite pas à la simple analyse dynamique de code en mémoire mais permet de faire de l'analyse statique, de la vérification par déduction, tests... [20]

Coccinelle Contrairement à *Frama-C*, *coccinelle* assiste le développeur dans la modification de code. Le point intéressant de cet outil réside dans la méthode pour détecter les erreurs. Le fonctionnement de cet outil se décompose en plusieurs étapes. Tout d'abord, *coccinelle* analyse les patchs modifiant le code. Puis, la modification est traduite dans des modifications isomorphes. Après le code C est traduit en un graphe de contrôle des flux (GCF) ainsi que les expressions qui sont traduites en logique temporelle arborescente (LTA). En appliquant les règles de logique temporelles sur le graphe de contrôle, il est possible de détecter des erreurs. Enfin, la modification du patch est appliquée sur les erreurs.

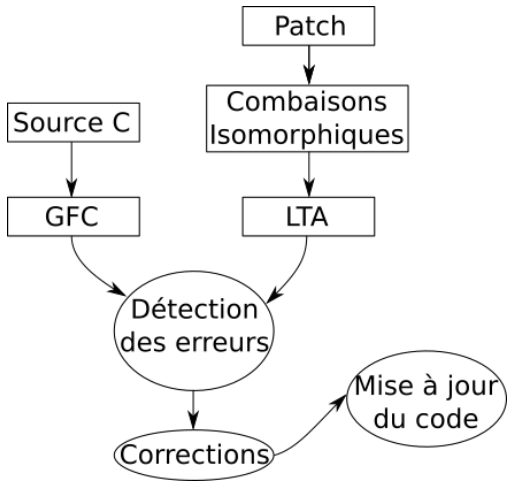

FIGURE 4 – Étapes de fonctionnement de coccinelle [21]

1.2.3 Correction automatique

Les outils d'aide à la correction, se situent dans le cadre de la réalisation du programme par un être humain. Cependant, il existe des mécanismes qui lors de la compilation ou de l'exécution vont appliquer des règles de sécurité. Nous décrirons ici uniquement trois mécanismes.

Compilation Certains compilateurs assurent preuves à l'appui que le code compilé aura le même comportement que le code source. Cette sûreté permet d'assurer que le programme aura bien le comportement souhaité mais aussi d'assurer une certaines sécurité. Par exemple, les deux compilateurs que nous allons présenter *compCertX* et le compilateur pour le langage *Rust* sont des compilateurs "threadsafe" [22]. Cela indique que ces compilateurs compilent du code qui ne permettra pas de faire des "races conditions". Il ne sera pas possible d'empêcher la mise en place de failles basées sur ce type d'erreurs. Cette méthodologie de vérifier la compilation se situe à la limite entre la protection *apriori* et *in tempore*. En effet, le compilateur s'assurera que des propriétés (comme la "thread safety") soient respectées dans le programme compilé.

Noyau Après avoir vu comment nous pouvions faire respecter des propriétés sur des programmes compilables nous pouvons nous demander comment les faire respecter sur des programmes qui ne peuvent pas être compilés. À ce niveau intervient la notion de noyau. En effet, ce dernier va assurer les échanges entre les programmes et les drivers. Bien que son rôle premier soit de faciliter la communication inter-processus (IPC), la distribution des ressources (Ordonanceur, allocation de mémoire...) ou encore la distinction de privilèges, le noyau doit s'assurer qu'un programme ne prenne pas le contrôle et se donne un accès privilégié à des ressources auxquelles il n'a pas droit. Dans ce cas, il s'agit d'une violation de la politique de sécurité. Nous pouvons citer deux exemples de noyaux qui sont conçus pour assurer différentes propriétés : *PipCore* et *mC2* du système *CertikOS*.

CertikOS Ce système possède un noyau (*mC2*) qui est entièrement prouvé en *Coq*. Ce noyau confine les programmes en s'assurant que certains prédicats sont toujours respectés comme :

- pas de data race : aucune possible de faire une "race conditions" pour atteindre une information non autorisée
- pas d'injection de code : un programme ne peut se modifier en mémoire
- pas de buffer overflow : un programme ne peut sortir d'un espace mémoire alloué
- pas de calcul overflow : un programme ne peut pas manipuler des nombres trop grands pour un type donné

Aussi, ce système peut exécuter des machines virtuelles et garantir que l'isolation entre elles est complète.

uCos Bien que n'étant pas un noyau, cet outil en permet la génération. Il s'agit d'un *framework* pour créer des systèmes sur mesure et les vérifier automatiquement. La déclaration du modèle du système ce fait dans un langage de haut-niveau avant d'être prouvé. Ensuite le code pourra être exporté dans un sous ensemble du langage C [23].

PipCore Ce noyau qui lui aussi permet d'exécuter des machines virtuelles, garantit l'isolation entre chaque partition mémoire alloué à des instances en mémoire. La preuve assure que toute portion de la mémoire est accédée uniquement par une partition qui en a le droit. Dans le cadre, de *PipCore* la partition concerne l'espace alloué à une machine virtuelle ou un processus. Cette preuve concerne donc la MMU (Memory Management Unit) gérée par *PipCore* qui a pour but de traduire les adresses virtuelles en adresses physiques.

Sel4 Ce microkernel est intéressant puisqu'il est entièrement prouvé au niveau fonctionnel. Cela indique que le code suivra exactement le comportement définit par la spécification. L'article [24] montre comment cette preuve a été mise en place et qu'elle est la méthodologie de développement. Contrairement, au différents systèmes montrés jusqu'à maintenant, ce système ne fut pas prouvé par l'assistant de preuve*Coq* mais par *Isabelle* qui est un autre assistant de preuve.

Raffinement Cette méthode consiste à décomposer successivement depuis un état abstrait vers un état concret. Dans le cas des langages de programmation cela consiste à ajouter des contraintes sur les valeurs d'un type de données.

$$n \in \mathbf{N} \mid n \geq 5$$

FIGURE 5 – Exemple de raffinement sur un entier

1.3 Trusted Computing Base

Positionement La *TCB* est positionnée aux endroits stratégiques du système. Dans le cas d'un *unikernel* multi-tâches, cette position peut être par exemple le changement de contexte qui dans le cadre de la sécurité est une étape critique.

Le positionnement dépend de l'organisation du système et de la hiérarchie des privilèges. Sur certains systèmes la *TCB* est l'interface entre le noyau et les utilisateurs alors que sur d'autres elle est juste une isolation mémoire. La mise en place d'une *TCB* est complexe puisque que le code est de confiance et ne doit pas permettre une violation des propriétés du système.

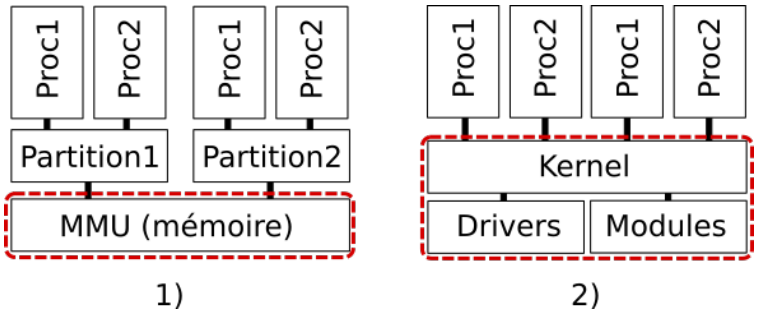

FIGURE 6 – Différentes *TCB* : 1. *PipCore* 2. *Linux*

Nous pouvons montrer un exemple très intéressant comme *Nemesis*. Ce système étant un *exokernel* il sépare clairement l'abstraction matérielle qui est gérée par les librairies systèmes, de la sécurité qui est exclusivement gérée par le noyau. La principale difficulté reste le positionnement. En effet, dans le cas d'un *exokernel*, la difficulté est de savoir quoi contrôler puisque ce sont les programmes qui, via les librairies systèmes accèdent directement au matériel.

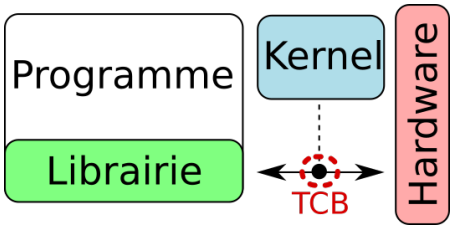

FIGURE 7 – Architecture d'un *exokernel* avec sa *TCB*

1.4 Système de contrôle d'accès

Dans le monde des systèmes d'exploitation, la protection des ressources passe par l'autorisation (l'identification et l'authentification) des acteurs à accéder aux ressources. Pour rappel, l'identification est l'action de déclarer son identité alors que l'authentification est l'action de prouver son identité. Il existe de nombreux moyens d'organiser la notion de droits, c'est à dire la notion entre ce qui est autorisé ou pas. Dans cette partie nous allons montrer différentes manières actuelles de gérer la notion d'autorisation entre acteurs, nous présenterons donc les principaux.

DAC *Discretionary Access Control* représente une forme très simple de droits. En effet, chaque utilisateur possède des ressources et indique comment ces dernières sont partagées avec les autres utilisateurs. Ce système ne permet pas la délégation de droits puisque ces derniers sont exclusifs au propriétaire effectif ou réel. De plus, en cas d'attaque par un cheval de Troie ou un utilisateur malveillant, la fuite d'informations est possible si l'utilisateur peut lire la ressource. Cependant, chaque ressource peut être contrôlée.

MAC *Mandatory Access Control* ce système de contrôle d'accès ne permet pas un contrôle d'accès précis à la ressource près comme *DAC*. Cependant, il permet d'éviter la fuite d'informations quel que soit le type d'utilisateur ou le type de programme. Après avoir défini des niveaux de sécurité, les flux entre ces niveaux sont limités. Nous pouvons citer un exemple, le modèle de Bell-LaPadula où les acteurs ne peuvent pas écrire dans des niveaux inférieurs au niveau actuel (*NoWriteDown*) et où les acteurs ne peuvent pas non plus lire dans des niveaux supérieurs (*NoReadUp*). Ce système a plusieurs points négatifs comme le fait que la décision de protection d'une ressource ne vient pas du propriétaire ou encore que dans certains cas (par exemple l'authentification d'un acteur), il est obligatoire de transgresser les règles concernant les flux (dans cet exemple la contrainte *NoReadUp*).

FIGURE 8 – Récapitulatif du système *MAC*, Bell-LaPadula

LBAC *Lattice Based Access Control* est un système de contrôle d'accès basé sur les treillis. Cela veut dire que tous les utilisateurs sont ordonnés et donc possèdent un supérieur et un inférieur. De cette manière, il est possible de définir des classes de hiérarchies et d'étendre ainsi un modèle *MAC* par exemple [25].

Dans la figure 9 si une ressource est au niveau (secret, proche) alors Alice et Bob pourront écrire dedans (si le modèle *MAC* est un modèle de Bell-LaPadula) comme des personnes du niveau public. En revanche, les personnes du niveau (secret,amis) ne pourrons pas écrire dedans.

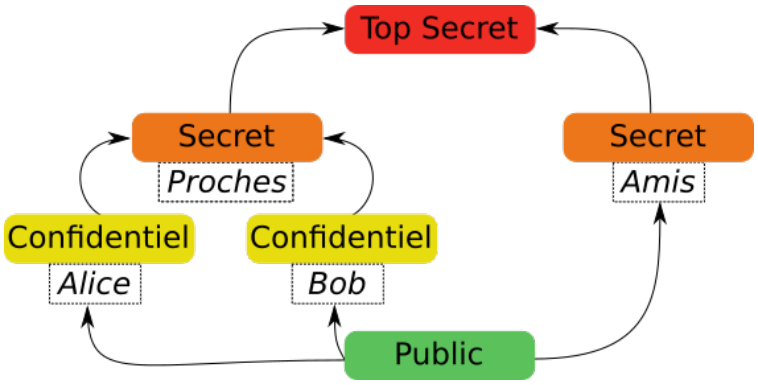

FIGURE 9 – Exemple graphique d'une organisation *MAC* sur un treillis

RBAC Le modèle *RBAC* permet d'affecter les droits non pas à l'utilisateur mais aux rôles qu'il endosse. Cela permet un système de gestion des droits dynamiques et une implémentation embarquée existe [26].

2 Thématiques de recherche

2.1 Mécanismes de sécurité

Liste de failles Dans le cadre du stage, une recherche exploratoire sur la sécurité des *unikernels* sera mise en oeuvre pour élaborer une liste plus complète des différents mécanismes de sécurité avec les avantages, les inconvénients et les limites. Le but de cette liste est de cartographier l'état actuel des mécanismes de sécurité. Le principe est de pouvoir mettre en avant de manière précise les différentes familles ainsi que leur liens.

Paternes Grâce à la liste des mécanismes, il sera intéressant de rechercher des paternes de failles ou d'attaques qui pourraient caractériser des familles de protection ou rassembler plusieurs familles. Cela permettra de présenter des scénarios classiques dans la sécurité des *unikernels*.

Limites Dans l'analyse des différents mécanismes, il serait utile de rechercher les failles matérielles qui pourraient survenir sur un système d'exploitation et en quoi les mécanismes de protection peuvent limiter le danger. En effet, cette partie amènera à préciser la limite des mécanismes, ce qui est intéressant dans le choix d'un de ces mécanismes.

Critères d'évaluation Toutes ces informations pourraient être utilisées pour voir si des critères d'évaluations spécifiques aux *unikernels* émergent. Cela pourrait être la base d'une recherche sur les bonnes pratiques de manipulation ou de conception spécifique aux *unikernels* mais aussi une aide considérable pour choisir un *unikernel* dans un cadre de sécurité défini.

2.2 Positionnement de la TCB

Après avoir clarifié, les limites des mécanismes de sécurité, il serait intéressant de voir si la position de la *TCB* peut ou pas améliorer la sécurité sans changer ses mécanismes. L'intérêt est

d'explorer ce qui dans un système doit être minimal pour assurer les prédicats souhaités. Ainsi, cette partie consiste à mettre en relation les mécanismes de sécurité et leur impact sur une *TCB*. Pour approfondir cette thématique, nous nous orienterons sur différents cas où il est nécessaire de positionner la *TCB*.

Modules De plus en plus de systèmes tendent à être modulaires et ce, dans le but de faciliter la maintenance ou la gestion des ressources. Nous répondrons à différentes questions comme par exemple : à quel niveau devrait être plaçé la *TCB* pour être efficace ? Ou encore existe-t-il un moyens d'interconnecter des modules qui ne sont pas de confiance avec le reste du système sans que la sécurité de ce dernier soit compromise ?

IPC Les messages inter-processus *IPC* sont vitaux aujourd'hui. Cependant, si ils permettent la synchronisation entre processus, ils représentent une menace pour le confinement inter-processus. Nous vérifierons quelles méthodes existent aujourd'hui pour lutter contre ces attaques et nous verrons si ces méthodes ont un impact sur la *TCB*.

Modélisation La recherche sur la position de la *TCB*, nous conduira à une définition formelle d'une *TCB*. Aussi, l'utilisation d'un formalisme pour décrire la *TCB* pourra être intéressant puisqu'à l'heure actuelle, il ne semble pas en exister.

2.3 Système de gestion d'accès

Dans le cadre des *unikernels*, la notion d'autorisation est vitale. Cependant, l'accès à cette notion est doublement critique. Il s'agit d'un accès qui doit être sécurisé, c'est-à-dire qu'une autorisation est toujours légitime. Aussi, le temps d'accès à cette notion d'autorisation doit être le plus court possible. Le but de ce thème n'est pas le résultat mais plutôt la méthodologie pour y arriver. L'idée étant de mettre en avant l'aspect méthodologique de la recherche dans un cadre pédagogique.

Propriétés Ce système devra respecter des propriétés. Voici les différentes propriétés souhaitées sur ce système :

- La simplicité afin d'être facile à implémenter
- Être résistant à un type d'attaque particulier (ici nous souhaitons nous pencher sur les attaques visant à faire fuir de l'information)
- Être facilement configurable (autoriser des problèmes classique comme la lecture d'un mot de passe sans être incohérent avec le modèle)
- Être le plus dynamique possible (par exemple, sur Unix le root est une entorse au modèle *DAC*)

Modélisation Ce système devra être formellement défini pour en déduire les propriétés minimales à son bon fonctionnement. Le but de cette partie est de sécuriser ce système de contrôle d'accès grâce aux connaissances apprises précédemment et par la preuve. Enfin, en fonction du temps restant nous pourrons terminer par une éventuelle implémentation.

3 Applications

3.1 Unikernels

La thématique de recherche dans ce domaine a pour but d'explorer la sécurité des systèmes embarqués sur des objets connectés ou sur des *Hyperviseurs*. Le principal intérêt est d'avoir des informations centralisées sur les risques encourus par les *unikernels* mais aussi les points à aborder lors de leur mise en place pour conserver une sécurité acceptable. Nous entendons par sécurité acceptable une méthodologie pour réduire un risque avec un impact fort (ou critique) vers un risque d'impact acceptable.

Mécanismes de sécurité La recherche sur les mécanismes de sécurité permet de recenser et de cartographier les différentes manières de sécuriser un *unikernel*. Cela permettra de vérifier si les critères de sécurité sont toujours d'actualité ou si ils doivent être mis à jours. Ces derniers pourrons servir à préciser ou à améliorer la sécurité dans les *unikernels* mais aussi dans les infrastructures. Les *unikernels* ayant pour vocation de se développer dans le *Cloud*, il est imaginable que dans le futur ils aient un impact important.

TCB La thématique de recherche sur le positionnement de la *TCB*, peut influencer la conception d'*unikernel* ou des systèmes comme par exemple *PipCore*. Ces observations peuvent aussi conduire à des recommandations pour la conception ou la configuration d'un *unikernel*. Aussi, la minimisation de la *TCB* est doublement essentielle puisqu'elle permet de simplifier la solution et elle permet de prendre en compte l'exécution de code non sûr, ce qui simplifie la construction d'un système d'exploitation. En effet, il n'est plus nécessaire dans ce cas de prouver l'ensemble des programmes. De plus, un système d'information ne peut pas être prouvé entièrement car il faudrait prouver tous les acteurs pouvant l'influencer.

3.2 Gestion de contrôle d'accès

Les propriétés définies précédemment peuvent s'appliquer à un système de fichiers pour un *unikernel*, pour gérer des utilisateurs sur une *ROM*. Le but ici est plus pédagogique afin de montrer la mise en place de la méthodologie pour présenter une idée, la formaliser, la démontrer et enfin peut-être l'implémenter. Cette implémentation sera plus dans le but de montrer le fonctionnement ou les performances plutôt que d'en faire une implémentation fonctionnelle.

Conclusion

Dans cette bibliographie, nous avons montré les différents moyens de sécuriser un programme sur toutes les couches logicielles, ce qui correspond aux domaines abordés par *unikernels*. Aussi, nous avons vu les mécanismes actuels qui permettent d'assurer qu'un programme soit plus sûr mais surtout plus sécurisé. Aussi, nous avons définit la notion de zone de confiance ce qui est très important lorsque la marge de manoeuvre pour la sécurisation est réduite. Enfin, nous avons abordé les différents moyens de contrôler l'accès à des ressources, ce qui dans le cas des *unikernels* correspond à un point important surtout si ce dernier fonctionne comme un *hyperviseur*.

Nous avons également soulignés les différents thèmes sécuritaires qui orienteront la recherche pendant le stage. L'importance des mécanismes a été énoncée ainsi que la réflexion qui peut être développée autour. De même, l'utilisation des mécanismes mais aussi de la position de la *TCB* pourra peut-être donner une vision plus claire des principaux moyens de sécuriser un *unikernel*. La sécurité de ce dernier est à la frontière de deux mondes : le monde de l'embarqué et le monde du *Cloud*. Ces deux mondes ont des impératifs liés à la criticité de leur rôle. C'est en cette unification que les *unikernels* apparaissent comme une solution viable.

Bien que nous nous penchons principalement sur les *unikernels*, il ne faut pas oublier qu'il ne sont pas la solution parfaite. En effet, l'*exokernel* a par exemple l'avantage d'avoir une *TCB* réduite. Cependant, ce type de système n'a jamais été complètement exploré. Bien qu'il existe une implémentation, elle ne fut jamais terminée. L'*unikernel* vient de *libOS* déjà existante. Pour des raisons financières, il est plus simple de sécuriser les successeurs directs de ces librairies plutôt que de refaire de nouveaux systèmes comme par exemple des *exokernels*.

Références

[1] Anil Madhavapeddy and David J. Scott. Unikernels : Rise of the virtual library operating system. *Queue*, 11(11) :30 :30–30 :44, December 2013.

[2] Ebios (expression des besoins et identification des objectifs de sécurité), 2017.

[3] Carl Gebhardt, Chris I. Dalton, and Allan Tomlinson. Separating hypervisor trusted computing base supported by hardware. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*, STC '10, pages 79–84, New York, NY, USA, 2010. ACM.

[4] Anil Madhavapeddy and David J. Scott. Unikernels : The rise of the virtual library operating system. *Commun. ACM*, 57(1) :61–69, January 2014.

[5] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel : An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[7] Steven M. Hand. Self-paging in the nemesis operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 73–86, Berkeley, CA, USA, 1999. USENIX Association.

[8] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[9] An overview of mirage os, 2017. `https://mirage.io/wiki/overview-of-mirage`.

[10] Riot - the friendly operating system for the internet of things, 2017. `https://riot-os.org/`.

[11] Contiki : The open source os for the internet of things, 2017. `http://www.contiki-os.org`.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5) :164–177, October 2003.

[13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels : Library operating systems for the cloud.

[14] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX Association.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Rump kernels : No os ? no problem ! *;login : Usenix Magazine*, 39(5), October 2014.

[16] Jean-François Monin. *Understanding formal methods*. Springer, 2003.

[17] Wikipedia. Liste en compréhension.

[18] Haskell, an advanced, purely functional programming language, 2017.

[19] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9) :269–282, August 2014.

[20] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. *Frama-C*, pages 233–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[21] Gilles Muller (INRIA), Julia Lawall (DIKU), Jesper Andersen, Julien Brunel, René Rydhof Hansen, Yoann Padioleau, and Nicolas Palix. Coccinelle : A program matching and transformation tool for systems code.

[22] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos : An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 653–669, 2016.

[23] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. A practical verification framework for preemptive OS kernels. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 59–79, 2016.

[24] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4 : formal verification of an operating-system kernel. *Commun. ACM*, 53(6) :107–115, 2010.

[25] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5) :236–243, 1976.

[26] Jae-Deok Lim, Sung-Kyong Un, Jeong-Nyeo Kim, and ChoelHoon Lee. *Implementation of LSM-Based RBAC Module for Embedded System*, pages 91–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.