MirageOS : la quête d'un OS plus petit et plus sûr

Thomas Gazagnaire

Résumé

Les techniques d'analyse statique de logiciels ont fait des progrès conséquents en terme de passage à l'échelle ces dernières années et l'analyse de programmes "réels" (plusieurs centaines de millions de lignes de code) est maintenant chose courante dans l'industrie. Bien que très utile, ces analyses sont souvent partielles et se concentrent sur un aspect particulier du problème à analyser.

Afin d'aller plus loin il est nécessaire de prendre en compte le contexte d'exécution de l'application. Ces environnements d'exécution sont de plus en plus complexes : (*i*) ils sont découpés en couches d'abstraction indépendantes afin d'assurer l'isolation et le bon fonctionnement des applications dont ils sont responsables et (*ii*) ils sont développés par des communautés utilisant des outils et des méthodes distinctes, ayant un intérêt pour les méthodes formelles plus ou moins développé. Toute cette complexité fait que la vérification de tels systèmes est aujourd'hui impossible.

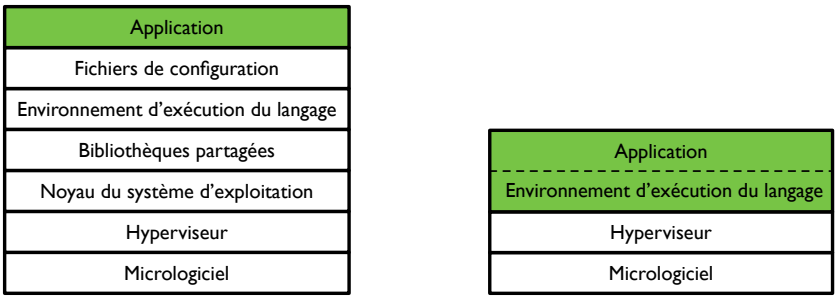
MirageOS, et plus généralement les *unikernels*, est une approche possible pour attaquer le problème de l'analyse et de la vérification de systèmes réels, environnement d'exécution compris. Cette approche consiste à (*i*) transformer un système d'exploitation en un ensemble de bibliothèques indépendantes écrites dans un langage de haut niveau (pour MirageOS nous avons choisi OCaml) – qui peuvent être certifiées et réutilisées dans différents contextes applicatifs ; et (*ii*) étendre la phase d'édition de liens aux couches basses du système d'exploitation pour générer statiquement des binaires comprenant le code minimal pour l'environnement et génère des systèmes avec une plus petite surface d'attaque, a priori plus facile à analyser et certifier.

1. Introduction

Les systèmes d'exploitation (*OS*) modernes (tels Linux, Windows, MacOS, etc.) sont structurés en différentes couches d'abstraction afin d'assurer au mieux l'isolation des différentes applications dont elles sont responsables. Ces systèmes sont chargés de gérer l'accès concurrent aux ressources physiques de la machine, en orchestrant au mieux les différentes applications et utilisateurs.

Avant de pouvoir entrer dans le détail de MirageOS dans la Section 4, il est important de comprendre pourquoi les systèmes actuels sont devenus si complexes. Pour cela, nous découpons arbitrairement cet environnement d'exécution en six couches logicielles, ordonnées par ordre croissant de privilèges de mode d'exécution, représentées dans la Figure 1 : la gestion des fichiers de configuration (Section 2.1), l'environnement d'exécution du langage (Section 2.2), les bibliothèques partagées (Section 2.3), le noyau du système (Section 2.4), l'hyperviseur (Section 2.5) et enfin le micrologiciel (Section 2.6) qui permet de programmer le processeur et le matériel. Un mode d'exécution plus élevé signifie qu'en général une couche donnée à le contrôle total de l'environnement d'exécution des couches plus basses. Le développeur exerce un contrôle fort sur les couches logicielles supérieures, ce qui lui permet d'utiliser des méthodes d'analyse, vérification ou certification de son choix ; la Section 2 montre comment ce contrôle diminue rapidement au fur et à mesure que le privilège du mode d'exécution augmente : sans contrôle ni connaissances sur le code qui est exécuté, il est impossible de mettre en place des techniques d'analyses classiques.

Afin de réussir à conserver ce contrôle, l'approche que nous poursuivons – détaillée dans la Section 3 – est celle des *unikernels* [10] : des systèmes ultra spécialisés, qui au moment de la compilation du projet, lient l'application à l'ensemble des couches logicielles nécessaires å son exécution. Comme illustré sur la Figure 1, le binaire produit est minimal : il contient uniquement



(a) système d'exploitation moderne (b) MirageOS

FIGURE 1 – Comparaison des différentes couches logicielles entre un système d'exploitation moderne et MirageOS.

le code dont il a besoin pour s'exécuter sur la plate-forme de déploiement choisie et il est plus sûr : le système d'exploitation monolithique est découpé en bibliothèques composables et analysables indépendamment – en particulier le développeur garde le contrôle dans le choix du code qui sera présent à l'exécution et dans le niveau d'analyse qu'il désire y faire.

Finalement, la Section 4 présente MirageOS, une implémentation en OCaml du concept des unikernels. Il se base sur (*i*) un ensemble de signatures de module qui modélisent les interfaces du système d'exploitation ; (*ii*) une centaine de bibliothèques qui implémentent ces signatures pour les divers pilotes et protocoles supportés et (*iii*) un outil qui génère le code nécessaire pour assembler une application en fonction de la configuration de son déploiement.

2. Le contexte d'exécution

Le but de cette section est de souligner que l'application, la partie qui nous intéresse réellement et que l'on a parfois passé des années à analyser et certifier est le haut de l'iceberg. En effet, chacune des couches de la Figure 1 présente des vulnérabilités, qui sont ici illustrées par des exemples pris dans la base de "Common Vulnerabilities and Exposures" (CVE) – une base de données en ligne d'informations publiques relatives aux failles de sécurité maintenu par MITRE [1]. Chaque section donne aussi un aperçu des efforts en cours pour fixer ce genre de problèmes à chacun des niveaux. Nous proposons, dans la section 3, d'apporter une solution globale – beaucoup plus radicale – à ce type de problèmes.

2.1. Les fichiers de configuration

Les fichiers de configuration, bien que ne formant pas une couche logicielle à proprement parler, sont souvent source de problèmes. Ces données sont lues au démarrage du programme et sont fournies par l'utilisateur : elles peuvent donc être malicieuses ou défectueuses.

D'une part, ils sont destinés à être facilement lisibles par un "être humain" : leurs formats sont ainsi généralement spécifiés informellement ou partiellement ; par exemple le format YAML dont la plupart des implémentations sont problématiques (Ruby : CVE-2017-2295, Python : CVE-2017-2810). D'autre part, afin de localiser les fichiers de configuration le programme doit, lors de son démarrage, interagir avec l'API POSIX d'accès aux variables d'environnement et au système de fichier qui peuvent avoir des comportements dynamiques complexes : par exemple lors de la phase d'expansion de variable, pour

1. https://cve.mitre.org/

trouver le répertoire racine de l'utilisateur (comme CVE-2004-0747), ou lorsque l'application n'est pas lancée avec le bon utilisateur (comme CVE-2002-1379).

Le choix du format et des défauts de configuration est normalement entièrement en contrôle du développeur de l'application ; les problèmes évoqués précédemment sont donc évitables.

2.2. L'environnement d'exécution du langage

OCaml, comme la plupart des langages compilés qui gèrent la mémoire de leur programme de manière automatique a un environnement d'exécution écrit en C – le rôle de cet environnement est d'instrumenter dynamiquement le programme afin de nettoyer régulièrement sa mémoire des données qui ne sont plus utilisées. Cet environnement est soit lié statiquement lors de l'édition de lien ou lié dynamiquement à l'exécution (comme pour la CLR) – ce qui apporte son propre lot de complications, voir la Section 2.3.

Comme tout programme écrit en C, l'environnement d'exécution peut comporter des vulnérabilités liées au dépassement de borne (CLR : CVE-2013-3134 et OCaml : CVE-2015-8869). L'environnement d'exécution du langage peut aussi accéder aux variables d'environnement, notamment pour configurer certains comportements (pour déboguer ou profiler un programme) – citons CVE-2017-9772 qui permet de faire exécuter du code arbitraire à l'environnement d'exécution OCaml.

Les environnements d'exécution de langage sont contrôlables dans une certaine mesure par un développeur d'application : dans ce cadre le choix du langage de programmation utilisé est clé. La petite taille et la (relative) simplicité de l'environnement d'exécution d'OCaml (représentation homogène et non typée des données à l'exécution, bibliothèque standard minimale, etc) – ainsi que des efforts de recherche récents tels *SecurOCaml* qui essaye d'identifier un sous-ensemble plus sûr de l'environnement d'exécution du langage – en font un bon choix d'environnement d'exécution.

2.3. Les bibliothèques partagées

L'environnement d'exécution du langage (Section 2.2 ainsi que les bibliothèques qui appellent des fonctions C vont, à un moment ou à un autre, faire appel à des primitives de la bibliothèque standard du langage C (*libC*). Cette bibliothèque est utilisée par la plupart des programmes présents sur la machine : afin d'éviter d'avoir à la dupliquer pour chaque exécutable, la *libC* est généralement partagée et chargée dynamiquement lors de l'exécution d'un programme. De plus la *libC* ayant une license GPL, c'est en général la seule manière de l'utiliser pour des applications dont la licence n'est pas compatible. Il existe ainsi un certain nombre de bibliothèques C qui sont toujours chargées dynamiquement (comme GMP qui est utilsée par `Zarith`).

La *libC* est une très grosse bibliothèque partagée : plus de 500 000 lignes de code C, avec son lot typique de dépassements de bornes (comme CVE-2015-0235) et d'accès (pas toujours sûr) aux variables d'environnement (comme CVE-2017-1000366). De plus, contrairement aux fichiers de configurations et à l'environnement d'exécution du langage, ces bibliothèques partagées sont totalement en dehors du contrôle du développeur de l'application : c'est en effet généralement le rôle de l'administrateur système de s'assurer que la dernière version des bibliothèques partagées est installée – c'est donc lui qui va décider d'une bonne partie de l'environnement d'exécution de l'application.

C'est la possibilité de redonner plus de contrôle au développeur vis à vis de l'environnement d'exécution qui a rendu populaire les approches "DevOps" basées sur les conteneurs logiciels (comme Docker). Ces technologies permettent de distribuer plus facilement des applications en contrôlant exactement quelles bibliothèques partagées sont installées en production. Parallèlement (et souvent en même temps) de nouvelles bibliothèques pour le langage C qui peuvent être liées statiquement sont proposées, telles `musl` qui a été popularisée par la distribution Alpine Linux.

2.4. Le noyau

Le système d'exploitation distingue deux modes de fonctionnement pour les programmes dont il a la charge : le mode utilisateur, qui impose une limite d'exécution à la plupart des applications écrites par un utilisateur et le mode noyau, qui permet d'ordonnancer les applications non privilégiées et de réguler leur accès aux ressources physiques de la machine (réseau, mémoire, disques, etc). L'interface entre le mode noyau et le mode utilisateur est un ensemble d'appels systèmes (ou *syscalls*) que les applications peuvent utiliser pour demander l'accès aux différentes ressources du système.

Les systèmes d'exploitation modernes sont des systèmes complexes : la dernière version du noyau Linux (4.14) comporte plus de 20 millions de ligne de code qui se partage entre pilotes de périphériques, gestion du système de fichiers, ordonnancement des applications et gestion des interfaces avec l'espace utilisateur. Ces systèmes exposent une API de plus en plus riche, soit plusieurs centaines de syscalls : 313 pour Linux[2] et autour de 500 pour Windows[3]. Ce large ensemble de syscalls couvre un spectre large de fonctionnalités ayant différents niveaux d'abstraction : par exemple, il est possible de demander un accès direct aux paquets qui transitent par une carte réseau (en utilisant des fonctions de lecture ou d'écriture sur une page de mémoire partagée entre l'espace utilisateur et le noyau), mais il aussi possible d'utiliser l'API des `socket` et d'avoir accès à un flot de données automatiquement encapsulé, par exemple, par le protocole TPC/IP. Ces protocoles réseau, qui forment la base d'Internet, sont généralement décrits à l'aide de spécifications informelles (les "Request For Comments" ou RFCs[4]) et chaque noyau (Linux, BSD, Windows, etc) implémente ces protocoles de manières subtilement différente. Comme ce code est exécuté en mode privilégié, la moindre vulnérabilité peut entraîner de lourdes conséquences, comme l'accès privilégié à la machine. C'est sans surprise que l'on retrouve ici aussi les vulnérabilités classiques des programmes écrits en C : dépassement de bornes (comme CVE-2005-0209) ou utilisation du pointeur `NULL` (comme CVE-2017-13686).

Pour un développeur d'application, il est assez difficile de choisir le noyau sur lequel son application va s'exécuter : seule la virtualisation (voir la Section 2.5) permet de spécifier la version exacte du noyau qui sera utilisée. Certains noyaux exposent des moyens de limiter le type d'appels systèmes que peut appeler une application : de manière assez grossière en utilisant les capacités POSIX ou de manière plus précise avec `seccomp`. Avec ces outils, le développeur peut annoter son programme afin de limiter son environnement d'exécution.

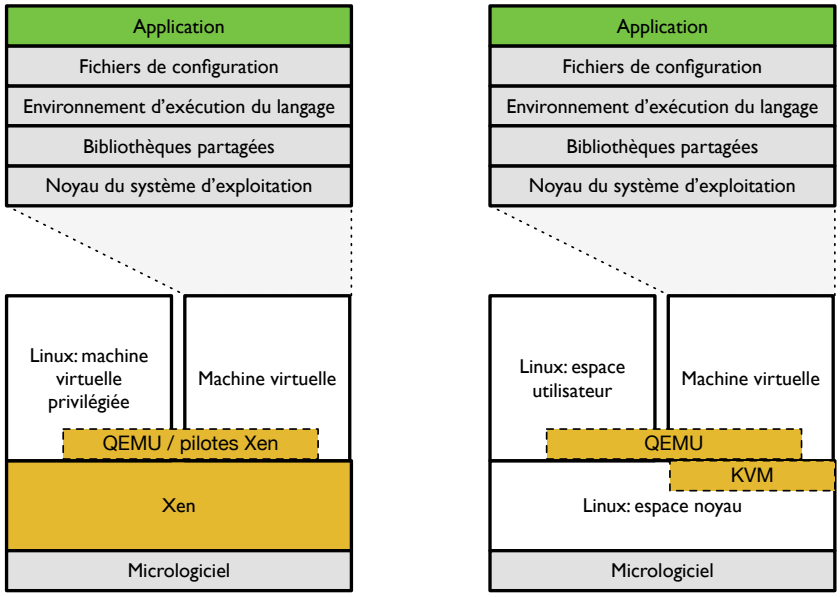
2.5. La couche de virtualisation a.k.a. "le Cloud"

Depuis une quinzaine d'années, la plupart des environnements de déploiement d'applications sont virtualisés : une machine physique arbitre l'accès aux ressources physiques (processeurs, horloges, disques, réseau, etc.) entre plusieurs machines virtuelles, isolées grâce à de nouvelles instructions disponibles sur les processeurs modernes (les extensions `VT-x` pour Intel et `AMD-V` pour AMD). Chaque machine virtuelle fait tourner son propre noyau qui gère ses propres applications.

Le logiciel responsable d'orchestrer un ensemble de machines virtuelle est appelé moniteur de machine virtuelle (VMM) ou *hyperviseur*. Il en existe plusieurs types, mais les plus répandus sont Xen [1] et KVM [6], qui sont utilisés par la plupart de fournisseurs d'hébergement public de machines virtuelles (a.k.a. le "cloud") comme Amazon ou Google, et VMWare qui permet à des entreprises de créer leur "cloud" privé. La Figure 2 compare l'architecture de Xen et de KVM. Xen (Figure 2.5) est une nouvelle implémentation des fonctions de base d'un système d'exploitation : gestion de l'orchestration (ici de machines virtuelles plutôt que de processus), gestion d'accès aux ressources physiques, etc. À l'opposé, KVM (Figure 2.5) est une extension du noyau Linux : les machines virtuelles

2. https://filippo.io/linux-syscall-table/
3. http://j00ru.vexillium.org/syscalls/nt/32/
4. https://www.ietf.org/rfc.html

sont ordonnancées et gérées de la même manière que des processus normaux – isolation matérielle en sus.



(a) Xen (b) KVM

FIGURE 2 – Architecture des hyperviseurs.

Bien que les processus d'isolation soient a priori différents, il est intéressant de remarquer que l'accès aux périphériques d'entrées/sorties est virtualisé et géré de manière similaire par l'interposition de nouvelles couches logicielles. Dans les deux cas, l'hyperviseur s'appuie sur un système d'exploitation qui s'exécute en mode privilégié : l'espace utilisateur classique de Linux pour KVM, une machine virtuelle faisant tourner un Linux pour Xen – permettant ainsi d'utiliser l'ensemble des pilotes de périphériques distribués avec Linux. Ce système d'exploitation privilégié sert d'intermédiaire entre les périphériques de la machine physique et les pilotes de périphériques des machines virtuelles, leur permettant d'échanger des données avec le monde extérieur. Si une machine virtuelle utilise des pilotes de périphériques classiques, le système d'exploitation privilégié doit lui les émuler. Pour Xen comme pour KVM, c'est QEMU (pour "Quick Emulator") qui est utilisé. Dans le cas de Xen, une machine virtuelle peut aussi utiliser des pilotes spécifiques pour un échange de données optimisé.

Sans surprise, comme tout projet complexe écrit en C, Xen, KVM et QEMU sont soumis aux vulnérabilités habituelles. QEMU, en particulier, a vocation à émuler de nombreux périphériques (utilisés par les machines virtuelles ou pas) : c'est un projet qui comporte plus d'un million de lignes de code en C – et qui inclut même une pile TCP/IP ! Ainsi, l'attaque VENOM (CVE-2015-3456) exploite un dépassement de borne dans un pilote de périphérique obsolète : il suffit de charger un pilote de lecteur de disquette dans une machine virtuelle pour prendre le contrôle de la machine physique !

Plus rarement, ce sont des vulnérabilités dans les algorithmes d'isolation des hyperviseurs (Xen : CVE-2017-10912 et KVM : CVE-2010-0306) qui permettent d'exécuter du code arbitraire dans un mode privilégié. Dans tous les cas, une vulnérabilité à ce niveau de la pile logicielle donne en général un accès complet à la machine et permet de compromettre l'exécution de n'importe application.

Pour un développeur d'application il est vraiment difficile de contrôler la version de Xen, KVM ou QEMU utilisée en production : le seul moyen dont il dispose est de changer de fournisseur d'infrastructure virtualisée. Cependant, ces fournisseurs sont conscients des conséquences liées aux

vulnérabilités des hyperviseurs : Google s'assure par exemple qu'il y ait toujours deux couches de virtualisation pour chaque application qu'il héberge et a récemment annoncé CrosVM, [5] une alternative à QEMU pour l'émulation de pilotes. D'autres initiatives pour remplacer QEMU sont présentées dans la Section 3.3.

2.6. Le processeur et son micrologiciel

Les processeurs modernes ont leurs propres systèmes d'exploitation. D'une part, des processeurs spécialisés, initialement utilisés pour gérer le démarrage, la mise en veille et la supervision du processeur principal ; ils sont en activité même lorsque la machine est en veille. Jusqu'à très récemment, nous avions peu d'information sur les capacités exactes de ces systèmes ; nous savons maintenant qu'Intel y incorpore un système d'exploitation complet (basé sur MINIX [6], incluant même un pile TCP/IP complète pour exposer un serveur web permettant la mise à jour de son micrologiciel). D'autre part, les processeurs implémentent la spécification UEFI, un véritable système d'exploitation de taille comparable au noyau Linux, qui est lui aussi actif en permanence et tourne dans un mode privilégié par rapport au noyau du système d'exploitation sous-jacent. Ces micrologiciels contiennent eux aussi des vulnérabilités (par exemple CVE-2017-5689) qui permettent de gagner un accès privilégié à la machine en envoyant un paquet spécialement conçu au processeur.

Un développeur d'application n'a a priori aucun contrôle à ce niveau – même si certains essayent de remplacer le micrologiciel par le noyau Linux [7]. Un aspect particulièrement inquiétant de ce type de vulnérabilité est (*i*) qu'il n'est pas détectable par les couches supérieures et (*ii*) qu'il n'est pas possible d'effacer un logiciel malveillant qui refuserait de se mettre à jour : la seule solution serait alors de jeter la carte mère et de la remplacer.

3. Les unikernels ou la simplification poussée à l'extrême

La section précédente présente le fonctionnement d'un système d'exploitation conçu pour exécuter *plusieurs* applications appartenant à *plusieurs* utilisateurs. De plus, ces systèmes sont capables de faire fonctionner tout type d'applications et en particulier des applications dont le code source n'est pas disponible (ou n'a pas encore été écrit) au moment ou ces systèmes sont déployés.

Dans cette section nous nous intéressons, au contraire, à un cadre d'exécution beaucoup plus restrictif : une seule application, un seul utilisateur, un environnement de déploiement connu et l'ensemble du code source de l'application, du système d'exploitation et des fichiers de configuration disponibles au moment de la compilation. Bien que contraignant, c'est la seule solution qui permet, en pratique de regagner du contrôle sur l'environnement d'exécution. Dans l'industrie, cela se traduit par exemple par l'utilisation des services immuables – ou l'on préfère redéployer un nouveau système plutôt que de modifier un système en cours d'exécution, ainsi que par le mouvement "DevOps", ou c'est le développeur qui est responsable de la mise en production de son application. Finalement, dans l'univers du logiciel embarqué, il est courant d'avoir un contrôle vertical complet sur la pile logicielle utilisée.

L'approche des *unikernels* est constituée de deux axes : développer un système d'exploitation de manière modulaire et étendre la phase d'édition de liens.

5. https://chromium.googlesource.com/chromiumos/platform/crosvm/
6. http://blog.ptsecurity.com/2017/04/intel-me-way-of-static-analysis.html
7. https://www.youtube.com/watch?v=iffTJ1vPCSo

3.1. Un système d'exploitation vu comme un ensemble de bibliothèques

Développer un système d'exploitation de manière modulaire[3, 8] permet d'utiliser facilement ces briques logicielles dans différents contextes et d'appliquer les techniques modernes de développement logiciel (tests, certification, preuves, etc) à des composants qui sont habituellement difficiles à modifier et à maintenir. Il existe deux approches : découper des systèmes d'exploitation existants ou écrire de nouveaux systèmes à partir de zéro.

Découper des systèmes d'exploitation existants Une première approche consiste à rendre modulaire les systèmes d'exploitation existants.

— FreeBSD a fait un effort pour découper son noyau en bibliothèques indépendantes et il est possible de ré-utiliser ces composants dans différents contextes. Appelés "rump kernels" [5], ces bibliothèques permettent de compiler des applications C sans modifications majeures si le système de construction du projet est capable de gérer la compilation croisée : seul l'appel système `fork` n'est pas disponible.

— Le principal effort pour apporter plus de modularité au noyau Linux est le projet "Linux Kernel Library" (LKL) [8]. Pour le moment, ces travaux n'ont pas vocation à être intégrés directement dans le noyau ce qui pose des problèmes de maintenance. De même que pour la *libC* (Section 2.3), ce projet pose aussi la question de compatibilité de licences dès lors que l'on veut se lier statiquement aux bibliothèques de LKL (distribuées, comme le noyau Linux sous GPL).

— Microsoft a aussi commencé à découper le noyau Windows en bibliothèques indépendantes avec le project Drawbridge [12]. Ces bibliothèques ont par exemple été utilisées pour écrire une couche de compatibilité entre les noyaux Linux et Windows - ce qui a permit de porter SQL Server (qui utilise l'API Win32) vers Linux.

Écrire de nouveaux systèmes De nombreux projets ont tenté de réécrire tout ou une partie des systèmes d'exploitation avec des outils plus modernes. *FoxNet* [2] re-implémente la pile TCP/IP en SML en s'appuyant sur le langage de module et de foncteurs – MirageOS reprend cette approche et l'étend à l'ensemble du système d'exploitation (voir la Section 4). *seL4* [7] est une implémentation d'un micro-noyau formellement vérifié.

En pratique, les nouveaux systèmes d'exploitation ont un problème fondamental : ils n'arrivent pas à écrire suffisamment rapidement les pilotes des nouveaux périphériques disponibles sur le marché, ce qui les rend très vite obsolètes. Cependant, la virtualisation (Section 2.5) permet d'exposer une interface très stable pour les pilotes de périphériques virtuels : ces pilotes sont donc devenus une cible privilégiés pour les nouveaux systèmes d'exploitation, centrés autour d'un environnement d'exécution pour un langage donné. Ainsi, *MiniOS* qui était initialement un démonstrateur pour écrire une machine virtuelle utilisant les pilotes de périphériques virtuels de Xen, est au final devenu la base de la plupart des unikernels actuels. Les autres unikernels utilisent les pilotes FreeBSD rendus disponibles par rump kernel. Le Tableau 3 donne un aperçu des projets unikernels existants, classés par langage supporté, plate-forme de déploiement et la technologie utilisée pour les pilotes de périphériques.

3.2. Étendre la phase d'édition de liens

Le deuxième aspect important des unikernels est qu'ils étendent la phase classique d'édition de liens pour analyser plus de code statiquement, en particulier en incluant l'ensemble des couches décrites dans la Section 2. Si l'éditeur de lien utilisé est optimisant, il est capable d'éliminer le code mort et de construire une application dont la surface d'attaque est minimale : uniquement le code

8. https://github.com/lkl

Projet	Langage	Pilotes	Déploiement
MirageOS	OCaml	MiniOS & solo5	Xen, KVM
rumprun	C	rump kernel	Xen, KVM
HalVM	Haskell	MiniOS	Xen
ClickOS	C	MiniOS	Xen
IncludeOS	C++	MiniOS, solo5	Xen, KVM
DeferPanic	Go	rump kernel	Xen, KVM
LING	Erlang	MiniOS	Xen
OSv	Java	MiniOS	Xen
runtime.js	Javascript	custom	KVM

FIGURE 3 – Résumé des projets unikernels.

nécessaire est présent lors de l'exécution de l'application – en particulier les pilotes de périphériques réseau et disques inutilisés ne sont pas présent dans l'exécutable final. De plus, les paramètres de configuration sont disponibles au moment de la compilation, permettant d'évaluer partiellement l'application avant même son exécution.

L'artefact produit est exécuté comme un processus unique, avec un espace d'adressage unique : en particulier le code de l'application et le code du noyau vont partager le même espace d'adressage, d'où l'utilité de s'assurer que certaines classes de problèmes liées à la corruption mémoire n'aient pas lieu et donc l'intérêt d'utiliser des langages typés de haut niveau.

À noter : le processus d'édition de lien peut s'arrêter à plusieurs niveaux – avec MirageOS il est par exemple possible de choisir entre utiliser la pile TCP/IP du système d'exploitation hôte ou la pile TCP/IP fournie par la bibliothèque OCaml correspondante ; l'utilisateur peut choisir de spécialiser, au moment de la compilation, l'environnement d'exécution pour le déploiement de l'application : ceci nécessite un contrôle fort sur le processus de construction du projet – rumpkernel utilise la cross-compilation ; MirageOS a développé un outil en ligne de commande pour configurer la compilation du projet — la plupart des autres projets demandent à leur utilisateurs de modifier manuellement les scripts de compilation des projets à compiler.

Au final, les avantages de cette approche sont spectaculaires : le temps de démarrage d'un unikernel est de l'ordre de quelques millisecondes[9, 11] : c'est moins que le temps de retransmission d'un packet TCP, ce qui permet de démarrer des applications à la demande. D'autre part, comme des couches d'orchestration sont supprimées, les latences observées sont en général plus facile à prédire que sur des systèmes d'exploitation classiques, qui sont fortement optimisés pour le débit de transmissions. Les unikernels produisent aussi des binaires petits et qui utilisent peu de mémoire : un serveur DNS écrit en MirageOS mesure environ 200 kilooctets et utilise 10 mégaoctets de mémoire vive.

3.3. Sécuriser les moniteurs de virtualisation

La Section 2.5 a mis en évidence les nombreux problèmes liés à l'utilisation de QEMU comme composant central dans la gestion des entrées-sorties. L'évolution naturelle des unikernels à ce niveau de la pile logicielle consiste à générer un moniteur de machine virtuelle spécialisé pour la supervision de l'application considérée ; cette spécialisation inclus les pilotes de périphériques virtuels (dans la machine virtuelle) et réels (dans le système d'exploitation hôte). C'est l'approche suivie par Solo5 [15] développé par IBM et qui permet à MirageOS de générer, en plus d'un exécutable complet entièrement spécialisé comme vu dans les sections précédentes, un moniteur entièrement

spécialisé qui va s'exécuter dans le système d'exploitation hôte et dont la seule tâche est de superviser l'application donnée. L'interface exposée par solo5 est minimale et facilement portable : un moniteur solo5 peut superviser une application déployée sur KVM+QEMU, FreeBSD+byhve, KVM+moniteur spécialisé ou le microkernel certifié *Muen* [9].

4. MirageOS : un exemple d'unikernel écrit en OCaml

MirageOS est un environnement de programmation pour construire des applications système (réseaux, stockage) performantes et portables. Le code peut être développé et testé dans un environnement classique (avec Linux ou MacOS) puis compilé en un artefact complètement indépendant, entièrement spécialisé pour l'environnement de déploiement tel Xen ou KVM (ou directement sur des machines réelles). MirageOS est entièrement écrit en OCaml et comporte un écosystème riche de plus d'une centaine de bibliothèques. Celles-ci fournissent des outils pour développer des applications système, en particulier liées au réseau car MirageOS inclut des bibliothèques pour les protocoles TCP/IP, HTTP et TLS. La gestion de la concurrence est basée sur `lwt` et il n'y a pas de multi-tâche préemptif (car l'appel système `fork` n'est pas disponible).

4.1. Utiliser des signatures de module standardisées

Les bibliothèques de MirageOS partagent toutes un ensemble standardisé de signatures de modules, qui définissent comment les différents composants d'un système d'exploitation peuvent interagir. Ces signatures comportent en général un type abstrait qui stocke l'état du périphérique ou protocole modélisé, mais il n'expose pas de constructeurs pour ce type : ainsi chaque implémentation est libre d'exposer des constructeurs utilisant les paramètres dont elle a besoin.

Un exemple simplifié de l'interface `Mirage_net.S` qui modélise une carte réseau peut s'écrire de la manière suivante :

```
module type S = sig
  type t
  type buffer = Cstruct.t
  type macaddr = Maccadr.t
  type error = private [> `Unimplemented | `Disconnected]
  val pp_error: Format.formatter → error → unit

  val write: t → buffer → (unit, error) result Lwt.t
  (** [write nf buf] outputs [buf] to netfront [nf]. *)

  val listen: t → (buffer → unit Lwt.t) → (unit, error) result Lwt.t
  (** [listen nf fn] is a blocking operation that calls [fn buf] with
      every packet that is read from the interface. The function can
      be stopped by calling [disconnect] in the device layer. *)

  val mac: t → macaddr
  (** [mac nf] is the MAC address of [nf]. *)
end
```

Une application MirageOS est un foncteur qui utilise ces signatures de module et qui possède une fonction `start`. L'exemple suivant montre une application qui affiche la taille de paquets que reçoit l'interface réseau à laquelle elle est connectée :

```
(* unikernel.ml *)
module Main (N: Mirage_net.S) = struct
```

9. https://muen.codelabs.ch

```
  let start n =
    let print buf =
      Logs.info (fun l → l "[%d]" (Cstruct.len %buf));
      Lwt.return ()
    in
    N.listen n print
end
```

Une application plus complexe, telle que le site de MirageOS [10] peut avoir de nombreux paramètres, pour avoir accès, par exemple à plusieurs espaces de stockage de clé/valeur en lecture seulement (représentés par la signature `Mirage_kv_ro.S`) et une interface pour gérer un serveur HTTP afin d'exposer des pages web (représenté par `Cohttp_lwt.Server`) :

```
(* unikernel.ml *)
module Main (FS: Mirage_kv_ro.S) (TMPL: Mirage_kv_ro.S) (S: Cohttp_lwt.Server)
  = struct ... end
```

4.2. Un écosystème de bibliothèques composables

Les signatures de module forment la colonne vertébrale de MirageOS. Une centaine de bibliothèques, disponibles sur *opam*, implémentent ces interfaces. Ces bibliothèques sont de deux types.

D'une part, il existe des implémentations concrètes de ces signatures, accompagnées d'un ou plusieurs constructeurs pour les types abstraits. Historiquement, ces implémentations concernaient les pilotes de périphériques virtuels pour Xen, développés dans le cadre de *XenServer* [14]. Aujourd'hui, elles couvrent un large spectre de protocoles réseau et de stockage.

Par exemple un pilote de carte réseau qui implémente `Mirage_net.S` en lisant une interface réseau en mode "raw" possède cette signature :

```
include Mirage_net.S
val connect: string → t Lwt.t
(** [connect i] opens the interface [i] (for instance "eth0") in raw mode. *)
```

D'autre part, il existe des foncteurs qui prennent en paramètre une ou plusieurs signatures de modules de MirageOS. Ces foncteurs implémentent eux aussi des signatures de module de MirageOS, habituellement de plus haut niveau et incluent un ou des constructeurs pour les types abstraits de ces signatures. C'est le cas par exemple de la couche ethernet : elle est paramétrée par une interface réseau (`Mirage_net.S`) et elle fournit une abstraction du protocole ethernet (`Mirage_ethif.S`) ainsi qu'un constructeur :

```
module Make (N:Mirage_net.S) : sig
  include Mirage_ethif.S with type netif = N.t

  val connect : ?mtu:int → netif → t Lwt.t
  (** [connect ?mtu netif] connects an ethernet layer on top of the raw
      network device [netif]. The Maximum Transfer Unit may be set via the
      optional [?mtu] parameter, otherwise a default value of 1500 will be
      used. *)
end
```

Finalement, la plupart des bibliothèques écrites en OCaml pur (sans liens avec du code C) et qui n'utilisent pas le module `Unix` peuvent être intégrés facilement dans une application MirageOS– le

10. https://mirage.io/

Tableau 4 donne aperçu des bibliothèques disponibles lorsque des interactions avec le système sont nécessaires.

Domaine	Bibliothèques
Interfaces Réseau	tuntap, vmnet, rawlink
Protocoles Réseau	ethernet, ARP, ICMP, IPv4, IPv6, UCP, TCP, DHCP, DNS, TFTP, HTTP
Protocoles de Stockage	FAT32, Git, tar, AES-CCM, ramdisk, B-trees
Sécurité	x509, ASN1, TLS, OTR
Crypto	MD5, SHA1, SHA224, SHA256, SHA384, SHA512, BLAKE2B, BLAKE2S, RMD160, 3DES, AES, DH, DSA, RSA, Fortuna, ECB/CBC/CCM/GCM modes

FIGURE 4 – Aperçu des bibliothèques disponibles avec MirageOS.

Certaines de ces bibliothèques ont fait l'objet d'attentions particulières : c'est le cas du protocole TLS [4] qui a été testé de manière rigoureuse [4] et du format de stockage des B-trees dont le code est généré à partir d'une spécification écrite en Isabelle/HOL [13].

4.3. Un outil pour configurer le contexte d'exécution

Avant d'être déployée avec un environnement d'exécution spécialisé, une applications MirageOS a vocation à être développée comme une application classique. Il est donc courant de vouloir configurer l'application selon différents modes, par exemple :

— `mirage configure --target=unix --net=socket` configure un binaire Unix dont le trafic réseau utilise l'API des *sockets* exposées par le noyau, ainsi que le ferait une application classique. Ce mode est utile pour que les outils classiques de déboguage et de profilage puissent fonctionner (avec les limites habituelles liées à `lwt`).

— `mirage configure --target=unix --net=direct` configure un binaire Unix servant un trafic HTTP via une pile réseau écrite en OCaml, en utilisant l'interface réseau par défault (`tap0` pour Linux) en mode "raw" Ce mode est utile pour tester la pile réseau.

— `mirage configure --target=xen --net=direct` configure un unikernel qui pourra être exécuté par Xen et qui utilise une interface réseau virtuelle. C'est ce mode qui est utilisé lors de la mise en production.

Le lien entre les signatures de module standardisées et les différentes implémentations disponibles est donc fait par l'outil *mirage*. Cet outil prend en entrée une application MirageOS :

1. un fichier `unikernel.ml` contenant un foncteur `Main` ayant au moins une fonction `start` ;
2. un fichier `config.ml` contenant une représentation de ce foncteur et de ses paramètres ;
3. des options en lignes de commande pour sélectionner les paramètres de déploiement.

Afin de spécifier la forme de l'application MirageOS à compiler, le fichier de configuration utilise une représentation abstraite d'un sous-ensemble du langage des modules d'OCaml qu'expose la bibliothèque *libmirage* (Figure 5). Les signatures de foncteur sont déclarées en utilisant le combinateur `@->`, qui représente la composition des arguments et la fonction `foreign` permet de nommer un foncteur. Les applications de foncteur sont représentées par le combinateur `($)`. La fonction `register` permet d'enregistrer le module principal de l'application.

Considérons, par exemple, le fichier de configuration suivant :

```
(** mirage.mli *)

type 'a typ
(** The type of values representing module types. *)

val (@→): 'a typ → 'b typ → ('a → 'b) typ
(** Construct a functor type from a type and an
    existing functor type. This corresponds to
    prepending a parameter to the list of functor
    parameters. *)

type 'a impl
(** The type of values representing module implementations. *)

val ($): ('a → 'b) impl → 'a impl → 'b impl
(** [m $ a] applies the functor [a] to the functor [m]. *)

val foreign: string → → 'a typ → 'a impl
(** [foreign name libs packs constr typ] states that the module named
    by [name] has the module type [typ]. *)
```

FIGURE 5 – Représentation d'un sous-ensemble du langage des modules en OCaml.

```
(* config.ml *)
let main = foreign "Unikernel.Main" (network @→ job) in
register "console" [ main $ default_network ]
```

La bibliothèque *libmirage* expose chaque signature de module standardisée comme une valeur de type `typ` : dans l'exemple précédent `network` représente ainsi la signature `Mirage_net.S`. *libmirage* expose aussi une valeur de type `impl` pour chaque implémentation disponible, ce qui permet de configurer, manipuler et assembler des environnements d'exécution complexes au sein même du langage. De plus, *libmirage* expose un mécanisme standard pour déclarer des paramètres à passer sur la ligne de commande au moment de la configuration ou de l'exécution. Ainsi, la variable `default_network` dans l'exemple précédent est une valeur de type `impl` qui "lit" sur la ligne de commande le mode de déploiement de l'application et sélectionne les paquets *opam* à installer ainsi que les bibliothèques à lier à l'application finale.

Grâce à ces mécanismes d'introspection, *mirage* peut par exemple décrire toutes les options possibles d'un projet avec `mirage -help`, ou l'ensemble des modules liés dans un mode d'exécution avec `mirage describe`.

5. Conclusion

La complexité des environnements d'exécution généralistes de ces systèmes rend aujourd'hui leur analyse impossible. Nous avons montré qu'il était possible au contraire d'architecturer un environnement d'exécution ultra spécialisé autour de chaque application et que les bénéfices de cette approche en terme de contrôle de l'organisation logicielle permettent d'envisager une analyse et une certification de ces nouveaux artefacts. Cependant, il reste encore du chemin à faire afin de rendre MirageOS encore plus sûr : continuer l'effort de vérification formelle des bibliothèques disponibles ; et surtout certifier l'environnement d'exécution du langage OCaml, qui devient maintenant l'élément de sûreté critique de ce nouveau système. Finalement, il serait intéressant d'explorer l'utilisation de MirageOS (ou d'un système équivalent) pour remplacer les micrologiciels des processeurs.

Remerciements Un grand merci à Louis Gesbert, Frédéric Bour et Romain Calascibetta pour leurs retours constructifs et suggestions.

Références

[1] 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.

[2] Edoardo Biagioni, Robert Harper, and Peter Lee. A Network Protocol Stack in Standard ML. *Higher-Order and Symbolic Computation*, 14(4) :309–356, Dec 2001.

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel : an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, Colorado, USA, 1995. ACM.

[4] David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-Quite-So-Broken TLS : Lessons in Re-Engineering a Security Protocol Specification and Implementation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 223–238, Washington, D.C., 2015. USENIX Association.

[5] Antti Kantee. *Flexible Operating System Internals : The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, Aalto University, Otakaari 1, 02150 Espoo, Finland, 2012.

[6] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM : the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4 : Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[8] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7) :1280–1297, 1996.

[9] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu : Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, 2015. USENIX Association.

[10] 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. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[11] 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.

[12] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–304, Newport Beach, California, USA, 2011. ACM.

[13] Tom Ridge. A B-tree library for OCaml. In *OCaml Workshop 2017*, 2017.

[14] David Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. Using functional programming within an industrial product group : perspectives and perceptions. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 87–92, Baltimore, Maryland, USA, September 27–29 2010.

[15] Dan Williams and Ricardo Koller. Unikernel Monitors : Extending Minimalism Outside of the Box. In Austin Clements and Tyson Condie, editors, *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.