Revue de littérature
Mécanismes sous-jacents aux *containers*

Julien DENIS
Promoteur: Jean-Noël COLIN
Université de Namur, Belgique

25 Mai 2019

Résumé

Les *containers* constituent une forme de virtualisation "légère" à envisager dans certains cas de figure où les performances et l'économie de ressources sont déterminantes, typiquement dans le cadre de services *cloud* ou de systèmes embarqués tels les *smartphones*. Même si l'on en fait traditionnellement débuter la popularité avec l'essor de Docker dès 2013, il s'agit d'une technologie dont les fondamentaux se sont construits au fil du temps depuis les années 1980. Sans en retracer l'historique complet, cette revue de la littérature des six dernières années met en lumière les mécanismes sous-jacents participant effectivement ou potentiellement, à quelque niveau d'abstraction que ce soit (de solutions "clé en main" à des circuiteries matérielles en passant par des fonctionnalités intégrées au noyau), à l'exécution de *containers* sur un hôte physique. De nombreuses expérimentations, principalement consacrées à Docker et Android, sont ainsi réalisées par des chercheurs en vue d'en augmenter l'efficience ou le niveau d'isolation. La sécurité des *containers* est intrinsèquement liée au niveau d'isolation entre ceux-ci, de même que par rapport à l'hôte, et constitue l'une des faiblesses souvent évoquées comparativement aux machines virtuelles soutenues par un hyperviseur dans le cadre d'une virtualisation matérielle traditionnelle.

1 Notes typographiques

Il n'aura pas échappé au lecteur que le présent article est rédigé en français. Or les ressources documentaires en science informatique sont fréquemment rédigées en anglais. Nous avons choisi d'utiliser, dans la mesure du possible, une graphie italique pour représenter les termes d'une autre langue que le français, typiquement l'anglais et les locutions latines, mais non par exemple les marques ou les noms de produit.

Notons également que nous avons fait le choix de conserver la dénomination anglaise de *container* au détriment de sa traduction littérale française de "conteneur" pour éviter toute équivocité potentielle liée à la signification générique de ce terme, alors qu'un sens particulier y est attaché dans le cadre du présent article.

2 Introduction

Le terme de *container* provient de solutions de virtualisation légère, telles Docker, lesquelles connaissent une popularisation croissante ces dernières années. On parle également de *container-based virtualization*, *operating system-level virtualization* (OSLV) ou encore *lightweight virtualization*, soit "*a number of distinct user space instances, often referred to as containers, are run on top of a shared operating system kernel*" [54, p.1] ou encore "*a variety of techniques used to sandbox, constrain, or simply modify the resource namespace of a process or a group of processes*" [102, p.355].

Si l'on mentionne une virtualisation "légère", on peut raisonnablement imaginer que, par complémentarité, il existe une virtualisation "lourde". Celle-ci fait référence à la virtualisation que l'on connaît depuis plusieurs décennies [15] visant à exécuter plusieurs systèmes d'exploitation sous forme de machine virtuelle au moyen de ressources matérielles communes gérées par un *virtual machine monitor* ou *hypervisor*. "*Hardware level virtualization is defined as emulating a set of hardware, usually for running a full-fledged, sandboxed operating system, through a hypervisor*" [44, p.2].

La virtualisation à laquelle se consacre cet article est qualifiée de "légère" dans la mesure où les ressources consommées sont moindres (singulièrement la mémoire). En effet, le noyau de l'OS hôte est partagé par les invités. Ainsi, contrairement à la virtualisation lourde, il n'est pas nécessaire de recharger à chaque fois en mémoire un OS complet (y compris le noyau). Le démarrage d'un *container* est beaucoup plus rapide qu'une machine virtuelle. "*Booting and loading the operating system kernel is a time-consuming and costly process [...] so it takes only a few seconds to create a new container*" [12, p.146]. Un hôte physique peut donc virtualiser davantage de *containers* (virtualisation légère) que de *virtual machines* (virtualisation lourde) avant d'épuiser ses ressources matérielles disponibles (calcul, mémoire, stockage,...). "*We can easily build hundreds of containers on a single physical machine, and we can only build a few virtual machines*" [12, p.147]. Une plus grande performance est, d'une manière générale, ce qu'offre une solution OSLV par rapport à une solution VMM. "*By avoiding the overhead of additional abstraction layers, containers are able to achieve near-native performance and outperform VM-based systems in almost all aspects*" [17, p.1] Le développement d'application et surtout leur déploiement se trouvent facilités par les *containers*. "*Rather than manually installing individual components, or packaging them into a heavyweight VM, all of the required components for an application can be encapsulated in a lightweight container that can easily be deployed on any platform that supports the container technology*" [102, p.355].

Si cette approche technologique présente des avantages, elle comporte aussi son lot d'inconvénients. En l'espèce, cela se manifeste principalement à deux égards. Premièrement, puisque le noyau de l'OS hôte est partagé, cela implique *de facto* qu'il n'est pas possible de virtualiser un *container* basé sur un noyau différent. "*All applications running on the container uses the same system calls provided by the host system, whereas [...] hardware virtualization emulate the whole system - both the kernel and the system calls*" [44, p.3]. Deuxièmement, les mécanismes d'isolation d'une solution VMM sont plus aboutis que ceux d'une solution OSLV [70, p.1]. Considérant que la virtualisation peut se manifester à divers niveaux d'abstractions (ISA, HAL, OS, application), "*in general, when the virtualization layer is closer to the hardware, the created VMs are better isolated from one another and better separated from the host machine, but with more resource requirement and less flexibility*" [64, p.1]. Comme nous le verrons, bon nombre des articles cités en référence cherchent à identifier ou améliorer les mécanismes existant en vue d'augmenter l'isolation des *containers* entre eux mais également des *containers* par rapport au système hôte.

Précisons toutefois que ces deux types de virtualisation ne sont pas mutuellement exclusifs et peuvent parfaitement coexister sur un même hôte physique. Rien n'empêche par exemple d'utiliser un hyperviseur pour créer une VM dans laquelle une solution OSLV sera installée sur laquelle siégeront des *containers* [17, p.3]. "*Most cloud providers, e.g., Amazon Web Services, run containers inside VMs*" [70, p.1].

Nous proposons une revue de littérature consacrée aux fondamentaux de cette virtualisation légère au cours de ces six dernière années, c'est-à-dire les mécanismes sous-jacents du système (indépendamment de la couche d'abstraction où ils se situent) participant à l'exécution de *containers*, avec un point d'attention sur la sécurité (isolation). Il ne s'agit donc pas de comparer la virtualisation légère à la virtualisation lourde, apporter des statistiques de performances, énoncer des méthodes pour gérer et orchestrer la masse de *containers* potentiels que l'on pourrait trouver dans des architectures *cloud*,... Nous nous limitons au cas de figure atomique d'une machine

physique unique supportant l'exécution d'un ou plusieurs *containers*.

Les solutions OSLV actuelles ne sont pas apparues du jour au lendemain, mais résultent d'un aggrégat de mécanismes développés petit à petit au fil du temps. La structure de cet article suit essentiellement une approche *top-down* en commençant par présenter de manière générique en quoi consiste une solution OSLV (titre 3) avant d'en présenter les domaines d'applications courants (titre 4). Nous présenterons ensuite les solutions "en production" (titre 5) et les solutions "expérimentales" (titre 6). Linux semblant constituer une plateforme de prédilection pour l'implémentation de solutions OSLV, nous présenterons les principaux mécanismes dudit noyau potentiellement exploités pour concrétiser l'existence de *containers*. Précisons que ces mécanismes ont été intrinsèquement peu développés au sein des articles constituant la base de cette revue de littérature. La raison principale tient au fait qu'il s'agit de technologies probablement considérées comme "connues" car relativement anciennes (à tout le moins antérieures à 2014). Néanmoins, il nous apparaissait utile de les présenter quelque peu puisqu'elles constituent la base même de l'existence des *containers* sous Linux. Ce titre constitue donc pour partie une sorte d'excroissance à l'inventaire de littérature tel qu'obtenu selon la méthodologie présentée au titre 14. A la suite du titre relatif au noyau Linux, seront indiqués les mécanismes matériels potentiellement impliqués dans la gestion des *containers* relevés dans la littérature (titre 8).

Au terme de ces titres cascadant de sujets généraux vers des sujets particuliers, nous présenterons les observations réalisées en matière de sécurité (titre 9). Nous proposerons des perspectives futures et questions de recherche (titre 10) avant de conclure cet article (titre 11). Figurent ensuite une liste de sigles (titre 12) et d'éléments terminologiques (titre 13) utilisés, la méthodologie ayant guidé la constitution du *corpus* de cette revue de littérature (titre 14) et enfin la bibliographie (titre 15).

3 Représentation générique d'une solution OSLV

Comme on peut l'observer sur la figure 1, un modèle d'installation typique [54] vise une série de containers qui fonctionnent sur une seule machine hôte physique. Le *kernel* hôte est partagé entre tous les containers.

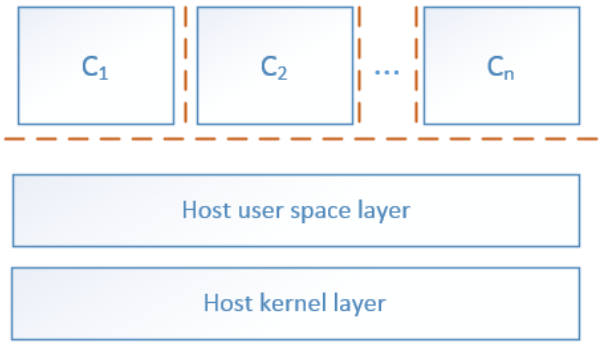

FIGURE 1 – Modèle OS-Level Virtualization selon [54]

Cependant, l'étendue de la couche relative à l'espace utilisateur de l'hôte varie selon le type d'installation :

- *Full*. On vise l'installation et la gestion d'un OS complet, sur laquelle une couche de gestion des containers vient se loger. Des ressources peuvent donc être partagées entre l'hôte et un ou plusieurs containers.
- *Light*. On vise la gestion allégée par laquelle la couche relative à l'espace utilisateur de l'hôte s'apparente davantage à une couche de gestion légère destinée à initialiser et exécuter les containers. Ce type d'installation serait plus sûr puisqu'il réduit la surface d'attaque dans la mesure où il n'y a pas de système hôte complet sous-jacent.

Dans chacun des types d'installation, un *container* peut également être de deux types :

— Application. Une seule instance d'application ou de service tourne à l'intérieur. On parle d'*application container* dans une installation *Full* ou de *Direct application/service setup* dans une installation *Light*.

— Système. Une installation dans l'espace utilisateur d'un OS entier. On parle de *system container* dans une installation *Full* ou de *Direct OS setup* dans une installation *Light*.

Des interactions sont toutefois fréquentes. Shan *et al.* classifient les interactions entre applications logées dans un *container* (*e.g.* enregistrement, notification, requête, réponse, authentification, lancement,...) en trois catégories [63, p.1220-1221] :

— Entre *containers*. Il s'agit de franchir des frontières normalement interdites par les mécanismes de virtualisation. Ce type d'interactions est subdivisé en deux sous-catégories :
  — *Container-Host*. Ce type d'interaction s'applique à toutes formes de technologie OSLV du fait de leur nature (partage du noyau et de l'environnement de l'hôte).
  — *Container-Container*. Sont envisagées la coopération entre applications logées dans des *containers* différents. Les échanges entre celles-ci seront normalement bloqués puisque cela déborderait des frontières établies. Une communication pourrait être opérée via le réseau mais cela dégraderait les performances (en considérant le cas visé ici de *containers* hébergés sur le même OS (*i.e.* le même hôte).

— A l'intérieur d'un *container*. Plusieurs applications peuvent résider dans le même *container*. Dans ce cas, il n'y a aucun problème de débordement de frontières puisque celles-ci se confondent. Toutefois, un problème potentiel réside dans la séquence de démarrage des applications (l'une appelle l'autre tandis que l'autre n'est pas encore en exécution). Or le fait de perdre du temps lors de cet amorçage nuit à l'un des avantages principaux que la virtualisation légère possède sur la virtualisation lourde.

— A l'intérieur d'un hôte. Cela vise les interactions originelles de l'environnement hôte et ne dépend donc pas des technologies de virtualisation.

4 Domaines d'application

D'une manière générale, les *containers* offrent une solution aux problèmes suivants : reproductibilité, portabilité, environnement de développement, intégration continue, flexibilité, isolation [69]. Notons que la portabilité diffère de la migration dans le sens où la première est une abilité, un prérequis pour la migration, tandis que celle-ci vise le déplacement lui-même [51, p.4].

Selon Loukidis-Andreou *et al.*, "*containers incur significantly less over-head than VMs, since they run as user-space processes on top of the kernel, which they share with the host machine. Moreover, they provide the ability to enclose application components in lightweight units, simplifying their distribution and deployment. As a result, large-scale applications can be managed in an automated manner*" [36, p.1561].

Selon Shan *et al.* "*OS-level virtualization is applicable for the applications that require both high performance and good isolation, including intrusion/fault-tolerance, server consolidation, high performance system, distributed hosting organizations like PlanetLab, as well as cloud computing in the future*" [63, p.1220]. Rappelons que la haute disponibilité est entendue comme autorisant une interruption de service de maximum 5 minutes par an, soit 99,999% de disponibilité [19].

L'un des milieux où les *containers* (tout comme les machines virtuelles d'ailleurs) sont mis à disposition généralement contre rémunération est le *cloud computing* et notamment le modèle IaaS. La manière de répartir les ressources est au coeur du métier de telles entreprises pour apporter à leurs clients les ressources nécessaires (CPU, mémoire,...) sous réserve de ce à quoi ils ont droit dans le cadre de leur plan tarifaire. Certains auteurs proposent de tenir compte des apports des la science économique en modélisant un système d'allocation basé sur la double enchère et l'algorithme du "recuit simulé" [12]. La migration de VM vise le déplacement d'une

VM d'un hôte physique vers un autre, soit de manière hors ligne ou en ligne. Mais il y a peu de littérature sur le sujet concernant les *containers* [22, p.215].

Les architectures logicielles conçues sous la forme de microservices poussent l'exploitation de *containers* qui sont bien adaptés pour ce genre d'usage [67]. "*Docker is an enabler for Microservices architecture and container based application deployment*" [67, p.806]. "*Containers can be launched within a second and are ideal for hosting event-driven microservices*" [70, p.1].

L'utilisation d'un *container* permet également d'empaqueter un logiciel avec les dépendances requises, le tout pouvant être exécuté sur n'importe quel ordinateur. Cela ôte toutes complications quant à l'exécution d'un logiciel de manière native sur un hôte quelconque (compatibilité, dépendances, numéro de version,...).

Les *containers* sont fréquemment employés, comme nous le verrons par la suite, dans le cadre de la prise en compte du BYOD visant "*permitting employees to bring personally owned mobile devices to their workplace, and access privileged company information and applications with these devices*" [13, p.1].

Mais on les trouve également dans d'autres domaines tels IoT, *devops*, bancs d'essais, VNFs, *fog computing*, *edge computing*,...

5 Solutions de virtualisation légère

Nous répertorions sous ce titre les différentes solutions OSLV en vue d'assurer une gestion locale des *containers*. Rappelons que la liste des solutions présentées n'est pas exhaustive mais énumère celles figurant dans les articles obtenus par le biais de la méthodologie exposée au titre 14.

Ces solutions sont considérées comme ayant un statut suffisamment abouti par leurs auteurs pour les mettre à disposition du public (gratuitement ou non) et assorties d'outils utilisateur afin de manipuler le cycle de vie des *containers*.

5.1 Chroot

Cette commande permet de modifier le répertoire racine au niveau du système de fichiers pour le *process* spécifié. Ce dernier n'aura dès lors plus de visibilité sur le reste de l'arborescence. Ce mécanisme aurait été ajouté à BSD par Bill Joy le 18 mars 1982, approximativement 17 mois avant la version 4.2 [24, p.131].

Notons que ce qui est entendu actuellement par la dénomination de *container* vise une isolation plus complète que ce seul changement de répertoire racine. Toutefois, cette commande *chroot* est fréquemment mentionnée comme l'ancêtre de la philosophie de la virtualisation légère, et il n'est pas rare de voir apparaître des dénominations telles *chroot-like* pour se référer à des solutions de type OSLV.

5.2 Jails

Une forme de "prison" a été introduite dans l'OS FreeBSD 4.0 en l'an 2000 [24]. L'objectif est de conserver la simplicité du modèle Unix (*root*) tout en rendant possible la délégation de certains privilèges (fonctions d'administrations) dans le cadre d'environnements virtualisés. Le cas d'utilisation évoqué par les auteurs vise les fournisseurs de service web qui emploient FreeBSD pour héberger les sites de leurs clients. "*However, these providers have a number of concerns on their plate, both in terms of protecting the integrity and confidentiality of their own files and services from their customers, as well as protecting the files and services of one customer from (accidental or intentional) access by any other customer. At the same time, a provider would like to provide substantial autonomy to customers, allowing them to install and maintain their own software, and to manage their own services, such as web servers and other content-related daemon programs.*" [24, p.117]

Pour atteindre ce résultat, une solution de partitionnement a été développée, sous le nom de *Jails*, en s'appuyant notamment sur d'autres mécanismes tels que *chroot*. En fin de compte, "*processes in a jail are provided full access to the files that they may manipulate, processes they may influence, and network services they can make use of, and neither access nor visibility of files, processes or network services outside their partition*" [24, p.117].

Par la suite, des améliorations ont encore été apportées à ce mécanisme en vue de renforcer l'isolation ou la flexibilité de gestion [54, p.7].

5.3 VServer

VServer est un projet semblable à Jails mais pour Linux, initialement publié en 2003 [50]. Dans ce jargon, on ne parle pas de "prison" mais bien de *virtual private server* (VPS). Pour fonctionner, il est nécessaire d'appliquer un *patch* au code source du noyau Linux. Ceci est considéré comme un inconvénient par divers auteurs dans la mesure où, d'une part, il ne s'agit pas d'une opération anodine et, d'autre part, cela complexifie la préparation des machines par rapport à une solution directement incorporée dans la branche principale de développement du noyau Linux [54, p.7].

La dernière version considérée comme stable du *patch* date de 2007 et était prévue pour la version 2.6.22.19 de Linux. Toutefois, le site *web* du projet continue de proposer des versions considérées comme expérimentales pour les versions récentes de Linux (4.9.159) [96], aussi bien au niveau du *patch* à appliquer au noyau qu'au niveau des outils proposés aux utilisateurs pour gérer les VPS.

5.4 Zones

Après avoir analysé Jails et VServer, Sun a publié en 2004 sa propre solution pour pour son OS Solaris (appartenant à présent à Oracle suite au rachat de Sun finalisé en 2010) considérant que "*at present they* [Jails et VServer] *lack the comprehensive support required for supporting commercial workloads. The barrier to entry for administrators is also high due to the lack of tools and integration with the rest of the operating system*" [52, p.242].

L'isolation fournie est basée sur un identifiant de zone affecté à un *process* destiné à limiter la visibilité de celui-ci aux frontières de la zone en question [54, p.7]. Il y a deux types de zones [51, p.3] :

- La zone globale voit toutes les ressources physiques et fournit aux zones non-globales l'accès à celles-ci.
- Les zones non-globales possèdent chacune leur propre système de fichiers, espace de nom de *process*, frontières de sécurité, adresses et *stack* réseau, comptes utilisateur et super-utilisateur.

Notons que le fournisseur *cloud* Joyent a développé SmartOS (commercialement présenté comme "la convergence entre les *containers* et les VM" [93]) par l'usage de technologies basées sur Solaris [61], plus précisément Illumos [83] qui en est dérivé.

5.5 OpenVZ

Ce projet a débuté en 2005 [28]. On y parle de *virtual environment (VE)* pour désigner les *containers*. A l'instar de VServer, cela vise Linux et nécessite d'appliquer des modifications au noyau. OpenVZ a été précureur [54, p.8] pour proposer une fonctionnalité *checkpoint* (entendue comme le fait d'enregistrer l'état d'un VE sous forme de fichier) *and restart* (consistant à faire redémarrer l'instance au départ de ce fichier) [43]. L'objectif étant notamment de pouvoir assurer une *live migration* de sorte que le *container* puisse être déplacé d'un serveur vers un autre. Cette fonctionnalité présente un intérêt pour assurer la haute disponibilité (soit 99,999% de

disponibilité ou, à l'inverse, une interruption de service de maximum 5 minutes par an [19]), la tolérance aux fautes et la répartition de charge entre plusieurs serveurs.

Le projet OpenVZ propose une distribution Linux prête à l'emploi (le noyau étant adapté et les outils utilisateurs disponibles) fondée sur RHEL. Une variante commerciale est supportée par Virtuozzo.

5.6 Docker

Construit au départ sur LxC (depuis la version 0.9 publiée en 2014 sur *libcontainer*, un moteur conçu avec le langage Go), Docker est considéré comme la solution la plus populaire par la littérature académique [17, p.3]. "*Docker is probably one of the most successful technologies for OS-level virtualization (or containerization), as it is being increasingly adopted by the major protagonists of the cloud computing arena, including Google and Amazon, though it is among the most recently proposed. Indeed, 92% of people surveyed by ClusterHQ and DevOps.com are using or planning to use Docker in a container solution*" [10, p.2]. Les raisons de la popularité qu'a engendré Docker et l'intérêt croissant pour les *containers* tiennent probablement au fait que les outils de gestion mis à disposition sont simplifiés et, de plus, il a proposé des dépôts de *containers* permettant de téléverser ou télécharger des images "prêtes à l'emploi" [18, p.2]. "*Docker has reinvigorated interest in lightweight virtualization by providing an easy-to-use interface for accessing the virtualization primitives built in to the Linux kernel, adding support for application image repositories, and leveraging an efficient copy-on-write file system for packaging application images with minimal space overhead*" [102, p.356]. Ou plus simplement "*Docker provides an easy to use application packaging and distribution mechanism, resulting in increased popularity, and several Docker-based container cloud offerings from Google, IBM, Microsoft, Joyent, amongst others*" [71, p.1].

Les applications sont empaquetées sous la forme d'une "image" qui consiste en un *snapshot* d'un système de fichiers en lecture seule. Ces images peuvent être récupérées depuis un dépôt public ou privé. Une telle image peut être exécutée et devient alors un *container*, entendu comme une instance dynamique de l'application [102, p.356]. Une image peut servir de base au lancement de plusieurs *containers* différents, tous indépendants. Les changements réalisés à l'intérieur d'un *container* sont persistants. Il est possible de partir d'une image et d'y apporter des modifications de manière automatisée via un fichier de configuration nommé *dockerfile* (lisible par un humain et rédigé sur un mode déclaratif [51, p.5]), le tout pouvant être téléversé sur le dépôt d'images mentionné précédemment, afin d'être téléchargé par la suite par un autre utilisateur qui bénéficerait du travail ainsi accompli. Une application et ses dépendances, enregistrées dans une image, peuvent être exécutées sur d'autres machines tout en s'assurant que le comportement reste consistant à travers les plateformes [17, p.3].

Docker utilise un système de fichiers par couche (dont plusieurs types sont pris en charge : UnionFS, AUFS, Overlay,...) de sorte que les changements apportés à l'image d'un *container* ne s'incorporent pas directement à celle-ci mais s'y superposent. Depuis l'intérieur du *container* toutefois, seul le résultat final des modifications sera visible et non toutes les étapes intermédiaires pour y parvenir ("*files in existing layers cannot be deleted or changed, instead they are hidden or overridden by an upper layer*" [18, p.2]). Cela offre des avantages (réutiliser des images de base communes, minimiser les téléchargements d'images,...) et des inconvénients (l'espace occupé par un *container* augmente à mesure des couches ajoutées, le temps d'accès est plus lent,...) [18, p.2].

Docker s'utilise par l'intermédiaire d'une interface en ligne de commande sur base d'une architecture client/serveur fondée sur le protocole HTTP (*Docker Remote API*), laquelle facilite la création d'outils tiers destinés à orchestrer les *containers*, à les gérer sous forme d'une ferme potentiellement composée d'un nombre élevé de *containers* (*i.e.* Kubernetes, Swarm,...). "*The Docker client can talk to the Docker daemon which creates, run and distribute Docker containers*" [67, p.806]. Une option de commande destinée à exécuter un *container* existe en vue de doter

celui-ci de privilèges proches de ceux que pourrait posséder l'utilisateur *root* sur le système hôte, quoiqu'il soit possible de les réduire via un ensemble de *capabilities* autorisées [102, p.356]. "*In order to access the kernel's lightweight virtualization primitives, the Docker daemon must run with root privileges. Each container process, which is forked from the daemon, initially runs as root and eventually relinquishes privileges leaving itself with the level of access specified by the parameters used to launch it*" [102, p.356]. Ils échappent donc aux contrôles imposés sur les utilisateurs standards de l'hôte. De plus, les utilisateurs accédant à l'API distante de Docker peuvent facilement obtenir un accès privilégié au système hôte [18, p.2].

Une approche de Wan *et al.* pour réduire la surface d'attaque potentielle d'un *container* compromis consiste à identifier les appels système dont il fait usage dans le cadre d'un fonctionnement légitime, puis de contraindre le *container* à cet ensemble en lui interdisant tout autre appel système [72]. L'identification des appels système légitimes est réalisé par le biais d'une analyse dynamique de l'exécution du *container* au moyen d'un outil de trace du noyau (en l'occurrence *sysdig*). "*Sandboxing, program analysis and testing are mature technologies. However, each of them has limitations : sandboxing needs policy, dynamic analysis needs executions, and testing cannot guarantee the absence of malicious behavior. [...] System call interposition-based sandboxing can guarantee that anything not seen yet will not happen*" [72, p.94]. Cette approche visant à réduire les appels système peut engendrer de faux positifs lorsqu'un *syscall* légitime survient alors qu'il n'avait pas identifié comme tel (car les tests dynamiques réalisés n'en avaient jamais requis l'appel). Mais des faux négatifs peuvent aussi se présenter pour les appels système autorisés alors qu'ils ne devraient pas l'être (par un usage détourné via injection de code d'un *syscall* normalement légitime, ou bien encore car le *container* analysé initialement pour en identifier les comportements était déjà malicieux dès l'origine) [72, p.100]. Enfin, une fois dressée la liste d'appels système considérés comme légitimes, celle-ci est appliquée sous la forme d'une politique *SecComp* qui est passée en argument de la commande lançant l'exécution du *container*. Dès lors, ce mécanisme de réduction de privilège peut difficilement être imposé à l'utilisateur si ce dernier a la possibilité d'exécuter lui-même un *container*.

Docker supporte tous les *namespaces* de Linux sauf ceux de *cgroup* et *user*, mais le mécanisme *cgroup* lui-même est bien supporté par Docker [34, p.420].

L'hôte et les *containers* appartiennent à deux différents domaines SELinux [10, p.3]. Afin d'éviter que tous les *containers* partagent le même type SELinux, Bacis *et al.* avaient proposé que les créateurs/mainteneurs d'images Docker introduisent des types SELinux spécifiques pour différents *processes* d'une manière qui soit transparente pour l'utilisateur final, à savoir via un *Docker Policy Module* spécifié dans un *dockerfile* englobé dans les métadonnées de l'image [5]. Une charge incomberait au *Docker Hub Registry* de vérifier que les différents *Docker Policy Modules* n'engendrent pas de conflits quant à l'intitulé des types entre deux images différentes. Notons que "*in order to prevent that precesses running in different containers access each other's resources, the containers domain is "partitioned" by means of multicategory security (MCS), another SELinux feature*" [10, p.3].

Docker est disponible sur les plateformes Windows et Mac OS, outre Linux naturellement. Or le principe même de la virtualisation légère repose sur un partage des primitives du noyau, alors que ce dernier est différent entre les trois OS mentionnés. Pour contourner ce problème, lors de son installation sur Windows ou Mac OS, Docker copie un noyau Linux (*i.e.* Alpine Linux [77]) et l'utilise pour exécuter les *containers*. Notons que les versions 3.3 à 3.9 de cette distribution Linux ont fait l'objet d'une vulnérabilité récente CVE-2019-5021 par laquelle on a observé l'existence d'un compte *root* doté d'un mot de passe à nul [1]. Quoiqu'il en soit, cette distribution est virtualisée dans le cadre d'une VM "traditionnelle" pilotée par l'application Docker [38] et supportée par VirtualBox jusqu'en 2016, depuis lors par Hyper-V sur Windows [11] et HyperKit, notamment basé sur le *hypervisor framework* natif des versions récentes de MacOS X ([37]). L'installateur de Docker conserve la possibilité de fonctionner avec VirtualBox pour le cas où un problème surviendrait par rapport à Hyper-V [80] ou HyperKit [79].

5.6.1 udocker

udocker est un outil d'intégration pour Linux publié en 2016 [97]. Il a été conçu afin d'être en mesure d'exécuter des *containers* dans un environnement utilisateur non privilégié ni même sur lequel Docker est installé [18]. Initialement, il s'agissait de permettre aux scientifiques d'échanger des logiciels développés dans le cadre de leurs recherches. L'objectif est en somme de fournir une fonctionnalité *chroot-like* mais sans utiliser l'appel système *chroot* qui exige un niveau privilégié. udocker utilise trois approches alternatives pour parvenir à cet objectif à savoir l'utilisation des *namespaces* utilisateur (à partir de Linux 3.19) ou du mécanisme PTRACE (dépend de la disponibilité de SecComp) ou encore du mécanisme LD_PRELOAD. Ecrit en Python, udocker propose une interface en ligne de commande similaire à celle de Docker. A la première exécution de udocker, celui-ci téléchargera les différents outils nécessaires. L'exécution de *containers* sans même devoir installer Docker est rendue possible par l'utilisation de runC, un outil développé par Docker et mis à disposition par OCI [90], qui propose notamment un mode de fonctionnement *rootless*, non privilégié. Par nature, l'utilisation de udocker ne pose pas de risque de sécurité pour l'hôte puisqu'aucun privilège administrateur n'est impliqué. Tout est confiné à l'ensemble de droits dont dispose l'utilisateur.

5.7 Cells

Cells est la seule (selon [54, p.8] en 2014) solution OSLV à source ouverte pour *smartphone*, fonctionnant en l'occurrence avec l'OS Android. Cells "*introduces a usage model having one foreground VP* [Virtual Phone] *that is displayed and multiple background VPs that are not displayed at any given time* [2, p.174]. L'objectif est de permettre le BYOD tout en segmentant les environnements d'exécution, typiquement privés et professionnels.

Cells introduit des *device namespaces* grâce auxquels les pilotes sont informés du *namespace* actif, en l'occurrence le *container* à l'avant-plan.

Cells est implémenté en effectuant la plupart des modifications requises dans la couche du noyau Linux, lesquelles ne seront vraisemblablement pas ajoutées dans la ligne principale du code source Linux [13, p.3].

Cette solution est soutenue par l'entreprise Cellrox [78], laquelle ne semble plus très active (la dernière mise à jour de son *website* remonte à fin 2015 et le Nexus 6, sorti en 2014, est présenté comme le dernier produit pris en charge).

5.8 Autres

Les solutions ou standards suivants sont apparus brièvement dans les articles sélectionnés sans faire l'objet d'une analyse plus approfondie. Leur mention par les auteurs était plutôt employée dans le cadre d'arguments comparatifs par rapport à leur propre solution.

LxC Le projet *Linux containers* date de 2008 et rend possible la virtualisation légère en utilisant seulement des outils situés dans l'espace utilisateur. LxC est donc un ensemble d'outils mis à disposition de l'utilisateur pour gérer les *containers*. Ainsi, seules les fonctionnalités directement incorporées à la branche principale du noyau Linux sont utilisées par l'intermédiaire des outils mentionnés. Dès lors, contrairement à VServer et OpenVZ, il n'est pas nécessaire d'appliquer un *patch* au code source pour modifier le noyau de base. Le projet est soutenu par Canonical. Selon le site *web* du projet[85], LxC se fonde sur les composants du noyau suivants :

- *Namespaces (ipc, uts, mount, pid, network and user)*
- *Apparmor and SELinux profiles*
- *SecComp policies*
- *Chroots (using pivot_root)*
- *Capabilities*

— *Control groups*

Rocket rkt est un moteur, dont le code source est ouvert [91], de *containers* pour Linux écrit essentiellement dans le langage Go. Il est développé avec un principe de sécurité par défaut selon ses auteurs, quoique celle-ci n'a pas fait l'objet d'analyse dans le cadre des articles constituant cette revue de littérature. "*Rocket architecture is simpler than Docker, does not have client server entity, but rkt binary in every node*" [55]. Ce moteur est capable de faire tourner des images Docker ou OCI.

Open Container Initiative L'OCI comporte actuellement deux spécifications par rapport aux *containers* : l'une relative à l'exécution, l'autre relative à l'image.

Windows Container Introduit en 2016 par Microsoft, ces *containers* doivent exécuter des *processes* de l'OS en plus de ceux dédiés à l'application en question (à cause d'une interdépendance complexe entre les bibliothèques de l'OS) [51, p.2] et ne sont dès lors pas si "légers". Il sont de deux types : "*Windows server containers provide application isolation through processes and namespaces isolation features. Hyper-V Containers encapsulate each container in a lightweight virtual machine : Kernel is no longer shared between the containers*" [51, p.3].

6 Expérimentations

Sous ce titre, figurent les expériences de recherche visant à répondre à un besoin spécifique.

6.1 Sur base de Docker

6.1.1 SCONE

Secure CONtainer Environment est une surcouche de sécurité sur Docker faisant usage de la technologie Intel SGX [3]. Son utilisation nécessite un pilote SGX pour Linux ainsi que, pour améliorer les performances, le chargement d'un module au noyau de celui-ci. Pour le reste, l'intégration avec Docker est transparente.

6.1.2 HYDRA

HYDRA est le nom donné à une modification expérimentale du code de Docker (environ 1300 LOC) visant à ôter la relation de filiation entre les *containers* et le *daemon* de Docker [71]. En effet, un *container* nouvellement créé devient un enfant du *daemon* Docker. Dès lors, au moindre problème à celui-ci nécessitant son redémarrage, tous les *containers* sont également redémarrés. Ce *daemon* constituant un SPOF, Verma et Dhawan ont découplé celui-ci des *containers* créés. Observant qu'un *process* orphelin était automatiquement adopé par le *process* du sommet de la hiérarchie, à savoir *init*, ils ont employé ce mécanisme pour supprimer le lien de filiation de sorte que le *container* devienne le frère du *daemon*. Toutefois, procéder ainsi empêcherait à celui-ci de communiquer avec les *containers*. Or Docker doit être en mesure de gérer le cycle de vie de ceux-ci. Dès lors, avant de créer le *container*, un petit *process* de monitorage est d'abord instancié, lequel créera ensuite le *container*. Ce *monitor process* servira d'intermédiaire entre le *daemon* et le *container*. Chaque *container* sera le fils d'un *monitor* différent. Docker peut donc connaître un souci justifiant son redémarrage, cela ne mettra plus en péril le temps de fonctionnement des *containers* qui en dépendent. Toutefois, qu'advient-il en cas de problème au *monitor* ? "*While the monitor process may itself be terminated inadvertently, it would affect just a single container. In the event of a monitor crash, the orphaned container is adopted by the INIT, and continues to function normally without any downtime. However, the daemon has no way to control/interact with such a container. HYDRA's daemon overcomes this problem by (a) maintaining a list of pid for both the monitor and the container, and (b) periodic polling for the monitor processes*

in a dedicated thread. When the daemon detects a live but orphaned container (based on the pid associations), it forcefully takes down and reboots the container, thereby maintaining connectivity with the monitor and the container." [71, p.3].

6.1.3 Harbormaster

Harbormaster est une application jouant un rôle d'intermédiaire à Docker en vue de rendre possible l'application de politiques de sécurité relatives à la gestion de *containers* par les utilisateurs reconnus de l'organisation [102]. Il ne s'agit donc pas de se protéger directement contre des attaques extérieures, mais bien simplement de laisser la possibilité aux administrateurs informatiques d'implémenter le principe du moindre privilège aux utilisateurs légitimes des *containers*. En effet, le *daemon* Docker étant privilégié, un utilisateur pourrait exécuter un *container* qui disposerait d'accès *root* à l'OS de l'hôte. Un autre problème que tente d'endiguer Harbormaster est la tendance *ready-to-run*, à savoir la possibilité de récupérer et exécuter une image de *container* depuis des dépôts publics, lesquels proviennent de sources qui ne sont pas nécessairement de confiance et susceptibles de servir de vecteur d'attaque. Harbormaster est une application *proxy* dans le sens où elle intercepte les commandes envoyées au *daemon* Docker, les examine au regard des politiques définies sous la forme d'un fichier XML (l'action demandée par tel utilisateur est-elle autorisée ou non), puis les transfère au *daemon* ou pas. Les règles peuvent viser des opérations sur les images ou les *containers* (*i.e.* images instanciées en cours d'exécution).

6.1.4 Docker-sec

Docker-sec est une solution présentée par ses auteurs comme pouvant apporter "*the benefits of a MAC system, without having to deal with the complexity of maintaining it*" [36, p.1562]. Il s'agit essentiellement de définir de manière automatisée des profils de sécurité AppArmor à tous les niveaux du cycle de vie des *containers*. Cela passe par une analyse statique des paramètres passés, par exemple, lors des commandes *docker create*, *docker run* ou *docker info* (tels que les volumes, les utilisateurs, les privilèges, l'ID d'un *container*, son point de montage racine,...). Il est possible en plus de réaliser une analyse dynamique par l'intermédiaire d'une période d'entraînement durant laquelle docker-sec va collecter des informations sur le comportement du *container*, puis analyser le journal d'activités pour adapter le profil des règles définies initialement par l'analyse statique. Il est important que, durant cette période d'entraînement, seuls les utilisateurs autorisés et de confiance puissent accéder au *container* et ses applications afin d'éviter d'autoriser automatiquement des activités malicieuses.

6.2 Sur base de Android

6.2.1 PrivateDroid

PrivateDroid est une modification du *framework* Android pour permettre l'exécution d'applications tierces sur un mode privé [29]. Sous ce mode, dès la clôture de l'application, toutes les données qui auront été enregistrées localement par celle-ci seront détruites. En conséquence, à chaque lancement de l'application en mode privé, celle-ci considérera qu'elle est exécutée pour la première fois.

Toutefois, si les auteurs ont envisagé une approche d'implémentation sous forme de *container*, ils lui ont préféré l'utilisation et l'amélioration du mécanisme de *sandboxing* existant dans Android estimant que celle-ci offrait un meilleur équilibre entre sécurité et utilisabilité [29, p.30]. L'approche OSLV était considérée comme offrant le plus de sécurité mais au détriment de la consommation des ressources (stockage, mémoire, batterie) et de la facilité d'implémentation (requérant la modification de la couche noyau de l'OS) [29, p.29].

6.2.2 Condroid

En vue d'augmenter la sécurité des environnements mobiles notamment dans le cadre BYOD, la virtualisation légère est une piste de solution à condition de répondre aux contraintes suivantes : conserver les performances natives de l'appareil, conserver l'expérience utilisateur native, supporter plus de deux *personas* (identités/profils différents). Condroid porte LxC sur Android en vue de partager les ressources de l'appareil entre plusieurs environnements Android.

Contrairement à Cells, la plupart des modifications requises pour implémenter Condroid sont opérées dans la couche du *framework* Android (par exemple pour virtualiser l'affichage) et non le noyau Linux lui-même (bien que quelques adaptations soient tout de même nécessaires) [13]. "*Condroid outperform Cell (container based lightweight visualization) by introducing or modifying Android framework like introducing virtual binder drivers for IPC (inter process communication), modification to android windows manager, modification to inputs manager, modification to service manager, and introducing two different subdirectories (/data and /system) apart from native android tmpfs (temporary file system) and nonvolatilefs (nonvolatile file system) to achieve better storage and sharing system files*" [62, p.1292-1293].

Selon Hubert *et al.*, l'implémentation de Condroid est "*highly specific to a certain OS version and blends domain isolation with domain interaction, resulting in a weaker security model and a larger TCB*" [23, p.432].

6.2.3 VDroid

Au départ de l'architecture de Condroid, VDroid comporte deux services en plus [62] :

— KSM recherche des pages mémoire dupliquées entre les différents *containers* en vue des les fusionner et réduire l'espace mémoire consommé.

— KSMManagerService maintient un vecteur des *containers* et active le *daemon* KSM à certains moments précis : la création ou la suppression d'un *container*, ou encore le *switch* d'un *container* vers un autre.

L'objectif ultime de cette optimisation de l'espace mémoire vise la possibilité de créer un plus grand nombre de *containers* sur un même système.

6.2.4 SplitDroid

Tout comme Condroid, SplitDroid vise à porter LxC sur la plateforme ARM sous l'OS Android (cependant, aucun de ces deux articles ne fait référence à l'autre), ce qui implique une recompilation du noyau pour activer les fonctionnalités manquantes dans la version native du noyau sous Android. Si la base technologique OSLV est la même, l'enjeu diffère toutefois quelque peu dans la mesure où SplitDroid élabore son mécanisme d'isolation au coeur même de l'application mobile utilisée (*app*) [99]. "*Based on our observation, privacy leakage often occurs within one app, where private data in one component may be leaked through another component in the same app* [99, p.79]".

SplitDroid distingue un environnement d'exécution normal d'un autre qui est isolé. Au démarrage de l'appareil, un *container* constitué d'une copie du même *framework* Android que l'original est créé et sera utilisé pour confiner l'exécution des composants sensibles des applications. La détermination de ceux-ci est opérée par le développeur de l'*app* lui-même, typiquement par l'intermédiaire d'un héritage de classes Java. Toutefois, il peut y avoir plusieurs applications nécessitant l'utilisation de l'environnement sécurisé. Pour éviter une confusion entre composants sensibles de différentes applications, "*the isolated execution runtime maintains an identity table to enforce a signature-based check before isolated execution*" de sorte que "*SplitDroid only supports sensitive components from one app to run in the isolate execution environment at a given time*" [99, p.86].

Considérant que c'est au développeur de l'*app* de distinguer les parties de son application qui doivent être exécutées dans un environnement sécurisé, et considérant que l'essence même

de SplitDroid vise à empêcher une application de collecter des informations sensibles en son sein, nous nous interrogeons sur la raison qui pousserait un développeur d'*app* contenant un composant malicieux d'empêcher celui-ci d'atteindre son objectif en lui imposant de s'exécuter dans un environnement isolé. En effet, si ce composant y a été placé par le développeur, c'est bien un objectif malicieux qu'il poursuivait...

6.2.5 Architecture de Wessel et al.

Les auteurs Wessel *et al.* [73] ont développé une architecture OSLV pour Android prévoyant plusieurs groupes de mécanismes, tous focalisés sur trois aspects de sécurité (isolation, communication et stockage) :

- Gestion à distance. Un composant de gestion et contrôle de confiance permet d'établir une connexion sécurisée avec un serveur de gestion, que ce soit localement, via VPN si l'appareil dispose d'une IP publique, ou encore via un SMS chiffré.
- Contrôle d'accès. Contrairement au système Android de base où l'utilisateur *root* dispose d'un contrôle total sur le système, l'approche des auteurs est de supprimer systématiquement, par *container*, toutes les *capabilities* non nécessaires pour qu'une application fonctionne correctement.
- Réseau et téléphonie. Le filtrage et routage réseau peuvent seulement être configurés depuis un environnement de confiance (*e.g.* la négociation d'une clé de session symétrique).
- Stockage. Un chiffrement du système de fichiers complet est réalisé, la clé n'étant jamais écrite en clair dans un espace de stockage qui serait lisible par un attaquant sur l'appareil éteint. Elle se trouve dans un élément de sécurité protégé par un PIN. Par ailleurs, un contrôle de l'intégrité des fichiers est opéré sur base de leur signature (*hash*) devant correspondre à une liste blanche (ou ne pas se trouver dans une liste noire).
- Affichage et entrée utilisateur. Une interface graphique spéciale est proposée à l'utilisateur pour entrer un mot de passe de manière sécurisée. Le bouton d'allumage est utilisé pour faire apparaître cet environnement de confiance.

Une collection de certificats est gérée dans le cadre d'une architecture PKI en vue de protéger la confidentialité et l'intégrité des données situées sur l'appareil. On dénombre quatre types de certificats : *backend*, appareil, logiciel, utilisateur.

Les auteurs Wessel *et al.* considèrent un modèle de sécurité où les vecteurs communs d'attaque d'un OS Android peuvent survenir, hormis JTAG ou similaires car cela nécessite des modifications matérielles. Les auteurs considèrent également que l'attaquant n'est pas capable d'extraire les clés privées des éléments de sécurité. En respectant ces deux limitations, des tests réalisés par les auteurs ont montré l'efficacité du mécanisme de protection en prévenant l'installation d'applications inconnues (et donc potentiellement malicieuses), ainsi que des attaques utilisant l'exploit CVE-2012-6422.

6.2.6 Architecture de Huber et al.

Les travaux de Wessel *et al.* [73] constituent la base du travail de Huber *et al.* [23]. Ces derniers conçoivent une architecture où l'espace utilisateur est constitué de *containers* dont un seul (C_0) est en charge de la gestion de la sécurité. Ce *container* privilégié dispose d'une GUI de confiance (*e.g.* pour entrer un mot de passe). Il est le seul à réaliser des opérations cryptographiques, par exemple stocker les autres *containers* sur l'appareil de manière chiffrée. Les *containers* autres que C_0 (soit C_i) sont non privilégiés et isolés dans leur propre *namespace*.

Tous les *containers* (y compris C_0) sont isolés du *namespace* racine et réduits à des fonctionnalités minimales. Toutes les communications se font par des interfaces bien définies. Outre les frontières des *namespaces*, l'isolation et la protection des accès non autorisés sont assurées par des restrictions au niveau d'un LSM personnalisé, profil SELinux et *capabilities*. Les auteurs estiment qu'une compromission de n'importe quel *container* ne pourra pas porter atteinte à la

confidentialité du reste du système (au delà des frontières du *container* compromis). Même si C_0 est compromis, bien que l'attaquant contrôlera beaucoup de fonctionnalités, les données situées dans les autres *containers* resteront confidentielles. Les auteurs précisent toutefois que leur modèle d'attaque ne prend pas en compte les canaux cachés ou les attaques physiques avancées.

6.2.7 Gemini

Cette expérimentation vise la virtualisation d'un téléphone en avant-plan et de multiples autres en arrière-plan. La réalisation de la séparation entre ces téléphones virtuels se fonde sur LxC et fait essentiellement usage des *namespaces* et *cgroups*. "*At same time, there is just one VP is active in a smartphone, the others VPs is not active. The active VP is foreground that can use all system source and devices, but the unactivated VPs is background that can use share devices (Wi-Fi, sensor, GPS, SIM, etc.) and cannot use proprietary and closed devices (display, input, audio, camera device, etc.)*" [33, p.187]. La solution présentée par les auteurs améliore notamment le mécanisme de *switch* pour passer d'un VP à un autre. Essentiellement par rapport à l'architecture qu'ils avaient développé précédemment [100] selon laquelle l'affichage s'éteignait entre chaque transition de VP, ce qui occasionnait une gêne pour l'utilisateur.

6.2.8 Démarrage depuis un périphérique OTG

Une séparation des domaines fondée sur la TCB consomme des ressources système et batterie au *runtime*, ce qui nuit à l'expérience utilisateur, selon Xue *et al.* [98]. Ces auteurs proposent une manière alternative de disposer d'un environnement Android sécurisé par l'isolation physique des données stockées dans deux instances différentes. Parmi les six différentes partitions (systèmes de fichiers) requises au minimum par l'OS Android, trois d'entre elles seront dédoublées (*i.e.* présentes à la fois sur le support de stockage interne de l'appareil, mais également sur un périphérique OTG) : celles contenant le système (OS), les données et le cache (contenant des données, même temporairement). L'objectif étant de disposer d'un système Android natif hébergé sur l'appareil, mais également d'un système minimal et sain hébergé sur le périphérique OTG. C'est après avoir introduit ce dernier dans son appareil que l'utilisateur démarrera celui-ci afin de charger un environnement Android sécurisé minimal.

Pour y parvenir, après avoir créé les trois partitions indiquées précédemment sur le périphérique OTG et y avoir copié les fichiers requis, il est nécessaire de modifier la séquence normale d'initialisation de sorte qu'à la détection de la présence du périphérique OTG au démarrage, le système soit chargé depuis ce dernier et non depuis les partitions présentes dans le stockage interne de l'appareil. Toutefois cela ne peut se faire qu'après le chargement du noyau Linux. En d'autres termes, celui-ci sera le même (situé sur le support de stockage interne) quel que soit le système Android chargé par la suite (celui "normal" stocké en interne ou celui "minimal" stocké sur le périphérique OTG). Selon leurs tests, les auteurs observent environ 45% de délai supplémentaire au démarrage du système Android sur OTG que celui stocké en interne (soit respectivement 28,32 secondes contre 19,48).

6.3 Sur base de Windows 2000

Feather-weight Virtual Machine (FVM) est une architecture OSLV expérimentale développée pour Windows 2000 fondée sur l'application de *namespaces* "*which isolates virtual machines by renaming resources at the OS system call interface. Through a copy-on-write scheme, FVM allows multiple virtual machines to physically share resources but logically isolate their resources from each other.*" [101, p.1].

Shuttle est une modification expérimentale apportée à FVM, testée sur Windows XP et Windows 2000, en vue d'améliorer la gestion des interactions entre applications [63]. L'identification de celles-ci a nécessité un travail de rétro-ingénierie de la part des auteurs puisque le code source

de l'OS en objet, et singulièrement son noyau, n'est pas publiquement disponible. Ils ont conclu l'existence de quatre types d'interactions inter-applications :

— Les communications entre applications via un mécanisme IPC.
— L'invocation d'application par laquelle une application en démarre une autre.
— Les transferts de nom destinés à transférer des noms de ressources parmi des applications. Ces noms sont codés "en dur" dans les binaires applicatifs et peuvent donc échapper au mécanisme de renommage d'une solution OSLV.
— Les dépendances applicatives par lesquelles une application dépend d'une autre, cette dernière devant donc tourner en premier.

Dans ce prolongement, Shan *et al.* élaborent une méthodologie pour identifier et confiner les mécanismes IPC [66] sans perturber le fonctionnement des autres *containers* ainsi que de l'hôte. Pour ce faire, ils répartissent les dix-huit types de méthode IPC identifiés en sept catégories :

1. Port : LPC, RPC, COM/DCOM/COM+.
2. Pseudo fichier : *mail slot*, *pipe*.
3. Mémoire partagée : *mapping* de fichier.
4. Synchronisation : sémaphore, *mutex*, événement, *timer*.
5. Messages : message Windows, copie de données, presse-papier, DDE.
6. *Sockets* : *sockets*.
7. Fonctions dangereuses : fenêtre *find*, créer un *thread* distant, définir un *hook* de fenêtre.

Ensuite, trois principes généraux ont été déterminés :

1. Isolation : les communications IPC sont autorisées à l'intérieur d'un *container* mais s'arrêtent à la frontière de celui-ci. L'exécution de ce principe est différencié selon la catégorie de mécanisme IPC citée précédemment :
   — Catégories 1 à 4 : les appels système sont interceptés dans le noyau et une méthode de renommage est appliquée de sorte que deux *containers* ne partagent pas leurs objets IPC et ne peuvent dès lors pas communiquer.
   — Catégorie 5 : les messages sont bloqués dès lors que l'expéditeur et le receveur se trouvent dans des *containers* différents.
   — Catégorie 6 : une adresse IP exclusive est affectée à un *container* au moyen d'un alias associé à l'adresse IP du *container*.
   — Catégorie 7 : ces fonctions sont prohibées (créer un *thread* distant, modifier l'espace d'adressage d'un autre *process*, définir des *hooks* au niveau du système global, énumérer les fenêtres d'autres *containers* ou de l'hôte).
2. Objet global : les *processes* à l'intérieur d'un *container* peuvent accéder à n'importe quel objet global, à l'exception de ceux créés par un *process* d'un autre *container*. Pour y parvenir, une table est construite dans chaque *container* et enregistre tous les objets globaux créés à l'intérieur du *container*. Lorsqu'un *process* tente d'accéder à un tel objet, si celui-ci figure dans la table on lui donne accès à celui créé dans le *container*, sinon on renvoie le *process* demandeur vers l'objet global du système.
3. Objet hôte : les *processes* d'un *container* peuvent accéder aux objets IPC créés par un service système de l'hôte. Pour mettre ce principe en application, deux listes ont été construites :
   — Une longue liste contenant tous les objets IPC des services du système hôte.
   — Une courte liste contenant seulements les objets IPC utilisés récemment, ce qui permet de réduire le temps de recherche.

Zhiyong *et al.* ont décrit une méthode générique pour virtualiser les services Windows, dont certains servent à "augmenter" le noyau et fournissent des fonctionnalités critiques à d'autres services et applications [65]. La virtualisation légère sous Windows consistant à dupliquer une instance de chaque service dans chaque *container* n'est pas aisée pour plusieurs raisons :

— Certains services sont tels que le noyau interdit de les dupliquer.
— Pour dupliquer un service, il faut intercepter et modifier divers mécanismes (communications inter-*processes*, coopérations inter-services, manipulations du registre,...), ce qui est difficile compte tenu de la nature propriétaire du code et oblige à procéder à de la rétro-ingénierie.
— Certains services contiennent des noms de ressource qui sont codés "en dur" dans leurs fichiers binaires.

La méthodologie générique exposée [65] pour virtualiser les services Windows se déroule en quatre étapes :

1. Créer une nouvelle entrée dans la base de données du SCM avec un nouveau nom.
2. Instancier le *process* nouvellement créé en l'associant à l'ID du *container* adéquat, la procédure étant différente selon que le service est basé sur un type DLL (modifier une entrée dans le SCM) ou un type EXE (copier le fichier dans l'espace de travail du *container* et ajouter le chemin dans le SCM).
3. Maintenir les interactions inter-services existantes (via les interceptions de requêtes IPC et le renommage dynamique des ressources appelées via FVM) en tenant compte des particularités comme le fait qu'un *core-process* intéragissant avec l'hôte ne peut pas voir les requêtes IPC transformées.
4. Pour solutionner le problème des noms de ressources codés "en dur" dans les binaires de certains services, une interception des fonctions API concernées est opérée et le nom du service original visé (codé "en dur") est remplacé par le service virtualisé correspondant.

Les expérimentations réalisées par les auteurs concernent les OS Windows XP et 2000. Nous nous interrogeons sur la raison qui pousse des chercheurs à continuer de produire en 2016 des résultats de recherche relatifs à un OS qui est déprécié par son propre éditeur Microsoft et ne fait normalement (sauf exception, telle pour la vulnérabilité CVE-2019-0708 [86]) plus l'objet d'aucune mise à jour depuis 2014 pour l'un et 2010 pour l'autre (sans considérer les éditions "dérivées" telle Windows Embedded POSReady 2009 basée sur Windows XP et faisant l'objet d'un support étendu jusqu'au 9 avril 2019).

7 Mécanismes du noyau Linux

"*The Linux Operating System is probably the reference target for the development of container-based virtualization solutions*" [9, p.69]. Considérant la nature même des solutions OSLV reposant sur un partage de noyau et que celui-ci s'avère fréquemment Linux dans la littérature, il nous apparaît inévitable de relever les mécanismes principaux du noyau impliqués dans la mise en place d'une solution de virtualisation légère.

7.1 Chroot

Cet appel système change le répertoire racine du *process* appelant (pour autant qu'il dispose de la *capability* CAP_SYS_CHROOT) à celui spécifié en paramètre [88]. Ce répertoire racine sera dès lors utilisé pour tout chemin débutant par "/" (caractère oblique) visé par le *process* ainsi que ses descendants.

7.2 Namespace

Le mécanisme du *namespace* consiste en l'abstraction d'un type de ressources globales du système afin d'en restreindre la visibilité. Concrètement, un *process* qui a été affecté à un *namespace* donné ne verra que les ressources du système affectées à ce même *namespace*, mais il n'aura même pas connaissance de l'existence des autres. Ainsi, un *process* au sein d'un *namespace* percevra celui-ci comme étant "le" système global, du moins par rapport à la ressource

du système concernée (car plusieurs types de *namespace* peuvent exister, chacun segmentant la visibilité d'un type de ressource).

La mise en place de tels espaces de nom remonte au système d'exploitation Plan 9 : " *The foundations of the system are built on two ideas : a per-process name space and a simple message-oriented file system protocol*" [49, p.1]. Cela a commencé à être implémenté dans le noyau Linux en 2002 à partir de la version 2.4.19 avec un espace de nom dédié au montage du système de fichiers. Les types de *namespace* pris en charge par le noyau a augmenté au fil du temps [7]. Actuellement, Linux comporte sept espaces de nom [89], chacun abstrayant un type de ressources globales :

- *Control group* (voir le titre 7.3).
- IPC. Un espace de nom portant sur les communications entre les *processes* prend en charge deux types d'IPC : les objets System V et les queues de message POSIX.
- *Network*. Cet espace de nom vise les ressources système associées au réseau (interfaces, pile, table de routage, règles de pare-feu, ports,...).
- *Mount*. Cet espace de nom permet de déterminer l'arborescence de répertoires visible. Rappelons qu'un système basé sur Linux propose une arborescence de fichiers/répertroires découlant d'une racine unique. Un *mount namespace* donné permet de déterminer ce qu'un process qui y est associé considère comme étant l'arborescence du système global (des parties de celle-ci peuvent lui être ainsi totalement inconnues).
- PID. Chaque *process* possédant un numéro l'identifiant de manière univoque, il ne peuvent être deux à porter le même PID. Cet espace de nom rend possible l'isolation de grappes de *processes* afin que plusieurs d'entre eux puissent porter le même identifiant, à condition d'appartenir à des *namespaces* différents. Ce mécanisme est un des fondements importants rendant possible l'exécution de *containers*, chacun de ceux-ci pouvant se résumer dans sa plus simple expression à une grappe de processus.
- *User*. Cet espace de nom permet d'isoler les identifiants et attributs liés à la sécurité (typiquement les identifiants des utilisateurs et groupes). Ainsi, un utilisateur pourrait être non privilégié dans un *namespace* donné mais disposer de droits d'administration dans un autre (par exemple non privilégié sur le système hôte mais administrateur dans un *container*).
- UTS. Cet espace de nom permet de conférer au système des noms d'hôte et de domaine NIS différents selon le *namespace*.

7.3 Control group (cgroup)

Les *control groups* ont été implémentés initialement dans la version 2.6.24 (publiée en 2008), sur base de travaux de Google, puis ont évolué en termes de fonctionnalités proposées. Ils permettent l'organisation hiérarchique des processes en groupes, non pas pour les isoler les uns des autres comme via les *namespaces*, bien bien limiter leur consommation maximale de diverses ressources (mémoire, CPU,...), ainsi que les monitorer. Notons qu'il existe un *cgroup namespace* qui isole la visibilité qu'a le *process* sur les *cgroups*. Ce type de mécanisme permet de contrôler les comportements de groupes de processes, typiquement en limitant les ressources utilisables (CPU, mémoire, réseau, I/O,...). Dans le cadre du modèle de facturation des services *cloud*, cela peut être utilisé pour assigner les ressources payées par les clients aux *containers* dont ils sont locataires [17, p.3]. Cela permet également de bénéficier des performances promises en limitant la quantité de ressources mises à disposition parmi les différents clients du fournisseur *cloud* se partageant la même infrastructure matérielle [34, p.420].

7.4 Linux Security Module (LSM)

LSM est un *framework* servant à implémenter potentiellement n'importe quel modèle de sécurité (*i.e.* une manière de spécifier et appliquer des règles de contrôles d'accès) pour le noyau

Linux. Mais en lui-même, LSM n'implémente aucune mesure de sécurité particulière. Sa raison d'être provient du fait qu'il n'y a pas de consensus quant à la meilleure approche de sécurité (MAC, DAC,...) [74, p.604]. Dès lors, aucun modèle n'a été directement implémenté dans le noyau, mais bien LSM qui "*allows many different access control models to be implemented as loadable kernel modules, enabling multiple threads of security policy engine development to proceed independently of the main Linux kernel*" [74, p.605].

LSM assure un rôle de médiateur pour tout accès aux objets internes du noyau. "*LSM seeks to allow modules to answer the question 'May a subject S perform a kernel operation OP on an internal kernel object OBJ?'*" [75, p.4] Pour pouvoir répondre à cette question, LSM place un *hook* juste avant la demande d'accès et appelle la fonction d'un module référencé qui se chargera de déterminer la réponse.

7.5 Security-Enhanced Linux (SELinux)

Sur base de Flask [82], une architecture de contrôles d'accès de conception MAC réalisée par la NSA et la SCC, le projet a été porté sur Linux pour profiter au plus grand nombre [35, p.1] et intitulé *Security-Enhanced Linux* (SELinux), d'abord sous forme de *patches* puis implémenté sous forme d'un Linux Security Module [68]. "*When dealing with containers, the kernel Discretionary Access Control (DAC) is usually considered insufficient, due to the flexibility it gives to the subjects and the limited control it provides on the security policy. With Mandatory Access Control (MAC), subjects cannot bypass the system security policy*" [5, p.749].

Les contrôles d'accès sont définis sur base d'une politique mise en application par le système. Une politique est un ensemble de règles, dont chacune comporte trois éléments : utilisateur, rôle, domaine. Un utilisateur dispose de rôles (pouvoirs d'action) sur un domaine. Essentiellement, il s'agit de déterminer les interactions possibles par rapport aux ressources du système (utilisateurs, fichiers, répertoires, *processes*, mémoire, sockets, ports réseau,...). Pour ce faire, SELinux emploie un *label-based enforcement model* utilisable de deux manières [5, p.749] :

- *Type enforcement*. Par un label, on associe par exemple un *process* à un objet système (tel qu'un répertoire). Une politique SELinux détermine les actions permises, que le noyau fait respecter.
- *Multi-category security*. Un label assigné peut être spécialisé avec une ou plusieurs catégories. Cela permet de créer plusieurs instances du même type (label).

Considérant que toute l'architecture repose sur des labels associés aux ressources, l'enregistrement de ceux-ci par rapport aux fichiers est réalisé par le biais des attributs étendus du système de fichiers. SELinux est le modèle de contrôles d'accès intégré dans les distributions Linux de la famille Red Hat (RHEL, CentOS, Fedora) par exemple.

7.6 Application Armor (AppArmor)

Développé initialement par Immunix (sous le nom de SubDomain), AppArmor est une implémentation d'une architecture de contrôles d'accès de type MAC sous la forme actuelle d'un LSM. Il s'agit donc d'une solution concurrente à SELinux. A la différence de celle-ci qui se fonde sur des labels adjoints aux ressources, les règles de politique AppArmor s'appuient sur un *path-based enforcement model* [34, p.420].

Ainsi, c'est le chemin du programme qui va déterminer ses limitations, indépendamment de l'utilisateur qui exécute le *process*. "*In focusing on applications (at the expense of roles and data classification), AppArmor is built on the assumption that the single biggest attack vector on most systems is application vulnerabilities. If the application's behavior is restricted, the behavior of any attacker who succeeds in exploiting some vulnerability in that application also will be restricted. [...] If an AppArmor-protected application runs as root, and becomes compromised somehow, that application's access will be contained, root privileges notwithstanding*" [6].

AppArmor est le modèle de contrôles d'accès intégré dans les distributions Linux Suse, Debian et Ubuntu par exemple.

7.7 Capabilities

Sur un système Linux, l'utilisateur *root* doté de l'UID 0 dispose de tous les privilèges sur l'OS et n'est donc pas lié à une règle normale de type MAC ou DAC. " *While the simple UNIX privilege mechanism has more or less sufficed for decades, it has long been observed that it has a significant shortcoming : that programs that require only some privilege must in fact run with full privilege. The dangers of such a lack of flexibility are well known, as they ensure that programming errors in privileged programs can be leveraged by hostile users to lead to full system compromise*" [20, p.164].

Les *capabilities* constituent une réponse à ce problème en établissant un modèle de privilèges tel que les pouvoirs de *root* ont fait l'objet d'un découpage en 38 unités distinctes [87] :

— CAP_AUDIT_CONTROL
— CAP_AUDIT_READ
— CAP_AUDIT_WRITE
— CAP_BLOCK_SUSPEND
— CAP_CHOWN
— CAP_DAC_OVERRIDE
— CAP_DAC_READ_SEARCH
— CAP_FOWNER
— CAP_FSETID
— CAP_IPC_LOCK
— CAP_IPC_OWNER
— CAP_KILL
— CAP_LEASE
— CAP_LINUX_IMMUTABLE
— CAP_MAC_ADMIN
— CAP_MAC_OVERRIDE
— CAP_MKNOD
— CAP_NET_ADMIN
— CAP_NET_BIND_SERVICE
— CAP_NET_BROADCAST
— CAP_NET_RAW
— CAP_SETGID
— CAP_SETFCAP
— CAP_SETPCAP
— CAP_SETUID
— CAP_SYS_ADMIN
— CAP_SYS_BOOT
— CAP_SYS_CHROOT
— CAP_SYS_MODULE
— CAP_SYS_NICE
— CAP_SYS_PACCT
— CAP_SYS_PTRACE
— CAP_SYS_RAWIO
— CAP_SYS_RESOURCE
— CAP_SYS_TIME
— CAP_SYS_TTY_CONFIG
— CAP_SYSLOG
— CAP_WAKE_ALARM

Il est donc possible de conférer à un *process* un ou plusieurs privilèges déterminés sans nécessairement devoir l'exécuter sous le compte utilisateur *root*. Ainsi, en considérant par exemple l'objectif d'avoir la possibilité de définir un répertoire racine pour un *process*, seule la *capability* CAP_SYS_CHROOT est nécessaire. Par défaut les *containers* créés par Docker en possèdent 14 [34, p.420]. Dès lors, une escalade de privilège au sein d'un tel *container* ne permet d'obtenir "que" 14 privilèges au lieu de 38.

7.8 Secure computing mode (SecComp)

Intégré à Linux depuis la version 2.6.12, il s'agit d'un mécanisme du noyau qui contraint les *system calls* qu'un *process* peut invoquer. "*A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated*" [92].

"*Since the system calls provide entry points for the processes in one container into the host kernel, a malicious app may misuse system calls to disable all the security measures and escape out of the container*" [32, p.4]. Dès lors, le fait de restreindre le nombre d'appels système que peut invoquer un *process* permet donc de réduire la surface d'attaque. Par exemple, le fichier de profil SecComp par défaut de Docker inclut plus de 300 appels système [34, p.420]. Les travaux de Lei *et al.* [32] ou Wan *et al.* visent justement à "miner des sandboxes", soit constituer un profil SecComp suffisant à un *container* légitime pour fonctionner normalement [72].

7.9 MultiLanes

Une adaptation expérimentale du noyau [26] a été réalisée par Kang *et al.* en vue d'éviter l'engorgement des périphériques de stockage rapide (de type flash par exemple) sur les plateformes *many-core* [25]. Ce problème de contention vient du fait qu'une solution OSLV partage le même *I/O stack* (contrairement aux VM dans le cadre de la virtualisation lourde qui disposent chacune d'un *I/O stack* isolé).

En conséquence, notamment par le fait des accès concurrents aux structures de données partagées et l'utilisation de primitives de synchronisation, le débit sur le support de stockage diminue au fil des ajouts de *containers* en exécution sur l'ordinateur. Ce problème était caché avec les technologies mécaniques de stockage car le débit subissait une forte latence.

MultiLane est une modification du noyau Linux visant à fournir un *I/O stack* isolé à chaque *container* en virtualisant le périphérique de stockage, en virtualisant le pilote du périphérique de type bloc, en partitionnant les structures de données du VFS.

8 Mécanismes électroniques

Certaines programmations logicielles exploitent des circuiteries mises à disposition par les fabricants de composants matériels, typiquement le processeur. Quoique cela nécessite une prise en charge au niveau du noyau de l'OS, nous choisissons de les distinguer dans une section différente. L'existence de ces mécanismes matériels n'impliquent pas automatiquement leur prise en charge par l'OS.

8.1 TPM

Spécifiée par TCG [94], un *Trusted Platform Module* est une puce électronique destinée à fournir une base de confiance sur un équipement, essentiellement par l'identification de l'utilisateur et de l'appareil. Cette puce comporte des fonctionnalités cryptographiques et stocke les artefacts utilisés pour s'authentifier sur l'appareil (mots de passe, certificats, clés de chiffrement,...) [22, p.214]. En pratique, elle permet d'assurer que la plateforme est bien celle qui est attendue (un logiciel prévu pour fonctionner sur une plateforme donnée ne fonctionnera pas sur

une autre) et qu'elle est de confiance (elle n'a pas été corrompue entre deux démarrages par exemple).

Dans le cadre de la virtualisation lourde, le TPM est émulé pour chaque OS invité (VM). "*Each guest virtual machine accesses the unique emulated TPM, as it appears that there is a separate TPM for each platform and from the virtual machines' viewpoints they have their own TPM*" [22, p.215]. L'intégration de ce vTPM (*virtualized* TPM) dans un environnement virtuel basé sur les *containers* est proposé par ces auteurs sous l'une des deux architectures suivantes : le vTPM est placé dans le noyau de l'OS ou bien dans un *container* séparé.

8.2 SGX

En 2015, Intel a mis en place *Software Guard eXtensions* (SGX) [14], technologie qui, selon Intel, "*assure un chiffrement matériel de la mémoire pour isoler des parties de code et des données spécifiques d'une applications dans la mémoire. Intel SGX permet au code de niveau utilisateur d'attribuer des régions de mémoire privées (appelées enclaves), conçues pour être protégées contre les processus exécutés à des niveaux de privilèges supérieurs.*" [84]

"*This makes SGX a promising candidate for protecting containers : the application process of a container can execute inside an enclave to ensure the confidentiality and integrity of the data*" [3, p.689]. Selon les mêmes auteurs, afin de mettre en place des *containers* sécurisés utilisant SGX, il convient notamment de minimiser la taille de la TCB à l'intérieur d'une enclave afin de diminuer d'autant la surface d'attaque potentielle car toute vulnérabilité au sein d'une TCB compromettrait son intégrité.

Toutefois, Schwarz *et al.* sont parvenus à implémenter un logiciel malveillant, "*a cross-enclave cache attack from within a malicious enclave that is extracting secret keys from co-located enclaves*" [59, p.4], capable de contrecarrer les promesses de SGX [60].

9 Sécurité

Un aspect constamment (*i.e.* au fil des années et publications) mentionné dans la littérature scientifique concernant les solutions OSLV a trait à la sécurité, essentiellement au regard de l'isolation proposée par la virtualisation légère (OSLV), plus faible que dans celle de la virtualisation lourde (HVV) :

2015 "*The security and isolation of the containers is correctly perceived as the most critical point for container security*" [5, p.749].

2016 "*Arguably they offer weaker security properties than VMs because the host OS kernel must protect a larger interface, and often uses only software mechanisms for isolation*" [3, p.689].

2017 "*Containers lack same level of security controls which is used in virtual machines, because containers were developed with the different considerations in mind*" [51, p.5].

2018 "*Despite the success of container services, there always exist security and privacy concerns for running multiple containers, presumably belonging to different tenants, on the same OS kernel*" [17, p.1].

2018 "*Containers pose significant security challenges due to their direct communication with the host kernel, allowing attackers to break into the host system and co-located containers more easily than Virtual Machines [...] Containers were not designed with security in mind. [...] Containers are much more prone to attacks compared to VMs due to the absence of a hypervisor, which adds an extra layer of isolation between the applications and the host. Since containers and host share the same kernel, compromised or malicious containers can more easily escape out of their environment and allow attacks on the host kernel*" [36, p.1561].

2018 "*Containers provide less security isolation than VMs*" [70, p.1].

2018 "*Though it is a consensus that the container mechanism is not secure due to the kernel-sharing property, it lacks a concrete and systematical evaluation on its security using real world exploits*" [34, p.1].

Notons que la question de la sécurité des *containers* semble également une préoccupation par des figures de la consultance *business*. Ainsi, Gartner rapporte pour 2019 la sécurité des *containers* comme l'une des 10 priorités technologiques sur lesquelles axer l'amélioration de la sécurité [53].

Considérant que la base d'utilisation populaire des solutions OSLV siège sur Linux, les problèmes de sécurité posés sont expliqués par le fait que l'on devrait multiplier les contextes d'exécution des différents sous-systèmes pour différencier le *container* de l'hôte, mais certains ne sont pas considérés comme prioritaires et les autres impliqueraient des changement en profondeur du code source du noyau [17, p.1].

Nous constatons deux tendances dans la manière dont les auteurs traitent cette question dans le cadre de cette revue de littérature :

- Plutôt théorique par l'élaboration d'un modèle d'architecture des solutions OSLV sur lequel vont se livrer diverses menaces d'attaques.
- Plutôt pratique par l'exploitation de vulnérabilités connues en vue de détourner le système et observer dans quelle mesure cela affecte l'isolation des *containers*.

9.1 Modélisation d'attaques

Reshetova *et al.* considèrent que "*an important factor to take into account in the evaluation of the effectiveness of any virtualization technology is the level of isolation it provides. In the context of OS-level virtualization isolation can be defined as separation between containers, as well as the separation between containers and the host*" [54, p.1]. Ces auteurs développent donc un modèle d'attaque tel qu'exposé sur la figure 2. On considère que l'attaquant possède le contrôle total de certains containers, les autres containers restant sous le contrôle des utilisateurs légitimes. Ses objectifs peuvent être :

- Compromission de container : accès illégitime aux données des autres containers (C_k), MitM, affecter le flux de contrôle des instructions exécutées dans C_k.
- Déni de service : perturber le fonctionnement normal de l'hôte ou des autres containers C_k.
- Escalade de privilège : obtenir un privilège originellement non reconnu sur des containers C_j.

Comme on peut l'observer sur la figure 2, l'attaquant cherche à atteindre ses objectifs en passant par les mécanismes de communication ou les mécanismes de stockage. On peut dès lors déterminer un ensemble d'exigences de sécurité que chaque solution OSLV doit remplir :

- Séparation des *processes*. D'une part, prévenir l'utilisation des interfaces fournies par l'OS pour gérer les *processes* tels les signaux et interruptions (figure 2 - attaque 1a). D'autre part, prévenir l'utilisation d'appels système pour accéder à la mémoire d'un process via *ptrace* par exemple (figure 2 - attaque 1b).
- Isolation du système de fichiers. Prévenir l'accès illégitime aux objets du système de fichiers de C_k ou de l'hôte (figure 2 - attaque 2).
- Isolation des *devices*. Protéger les pilotes partagés entre différents *containers* et un hôte, car ils exposent des interfaces permettant d'exécuter du code dans l'espace noyau (figure 2 - attaque 3).
- Isolation IPC. Prévenir l'accès ou la modification de données transmises par C_k via différents canaux IPC tels les sémaphores, la mémoire partagée, les queues de messages (figure 2 - attaque 4).
- Isolation réseau. Prévenir les attaques via les interfaces réseau disponibles (figure 2 - attaque 5) par exemple pour espionner, modifier le trafic réseau de C_j ou de l'hôte, réaliser un MitM.

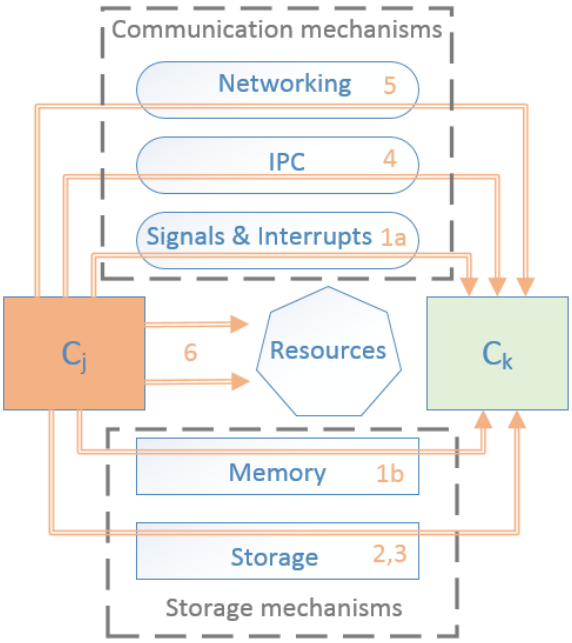

FIGURE 2 – Modèle de l'attaquant selon [54]

— Gestion des ressources. Prévenir l'exténuation des ressources physiques disponibles (espace disque, limites I/O, cycles CPU, bande passante réseau, mémoire) en fournissant un moyen de limiter les ressources disponibles pour chaque *container* (figure 2 - attaque 6).

Notons que ce modèle envisage l'isolation du *container* vers l'hôte ou des *containers* entre eux. Mais la perspective de protéger l'accès à un *container* depuis l'hôte n'est pas traitée. "*Tenants, however, want to protect the confidentiality and integrity of their application data from accesses by unauthorized parties - not only from other containers but also from higher-privileged system software, such as the OS kernel and the hypervisor*" [3, p.689].

Catuogno *et al.* rapprochent le modèle développé par Reshetova *et al.* du *Common Criteria Recognition Arrangement*, "*an international agreement that ensures the standardisation of criteria, methodologies, processes, documentation and recognition of the evaluation of security related products*" [9, p.72], ces deux modèles étant équivalents de leur point de vue dans le cadre du périmètre envisagé. L'intérêt de considérer cette équivalence tient à l'unification de deux perceptions (académique d'une part, industrielle d'autre part) d'un même problème et, par corollaire, ouvrir la possibilité de mutualiser les résultats obtenus.

Enfin, sur la question plus spécifique du réseau entre *containers*, Suo *et al.* ont réalisé une analyse différenciée des approches possibles à ce sujet (pas de réseau, mode bridge, mode *container*, mode hôte) et distingué les cas d'utilisation selon que les *containers* doivent communiquer alors qu'ils siègent sur le même hôte physique ou plusieurs [70]. Cependant, leur analyse empirique n'était pas indépendante d'une technologie particulière puisqu'ils fondaient leur observation sur les possibilités offertes par Docker. Par ailleurs, ils s'intéressaient non seulement à la sécurité (isolation) mais également à la performance. Leurs observations rapportent la difficulté de déterminer le mode réseau le plus approprié, mais suggèrent les pistes suivantes. "*To improve container network on a single host, communications should be performed through shared memory when possible and avoid packet copying between user space and kernel space as much as possible. To improve container networks across multiple hosts, the expensive packet encapsulation and decapsulation operations can be accelerated through hardware offloading*" [70, p.9].

9.2 Exercice d'attaques

Certains auteurs préfèrent inscrire leur approche dans une démarche pratique. Ainsi, Lin *et al.* ont collecté 223 exploits [81] susceptibles d'affecter une plateforme de *containers* [34]. Plus généralement, ils considèrent qu'un exploit susceptible d'affecter le noyau Linux peut atteindre la plateforme de *containers* également, puisque l'architecture OSLV utilise un noyau commun. Après avoir retiré les exploits redondants et inefficaces au moment de leurs tests, 88 exploits étaient encore opérationnels. Relevons au passage que l'expérience a été menée par ces auteurs sur des *containers* gérés par Docker 17.09.1-ce tournant sur une plateforme Linux Ubuntu 14.04, le tout faisant l'objet d'une configuration par défaut. Ils ont constaté que 56,82% des 88 exploits sont en effet capables de mener une attaque depuis un *container*. Une taxonomie à deux dimensions de ces exploits a été élaborée :

— Conséquence : fuite d'information sensible, prise de contrôle à distance, déni de service et escalade de privilège.
— Couche d'influence : application *web*, serveur, lib, noyau.

Ils ont observé que les mécanismes de sécurité du noyau (*i.e. capability, SecComp, MAC*) ont un rôle plus important pour prévenir l'escalade de privilège que les mécanismes d'isolation des *containers* (*namespace, cgroup*). Toutefois, les relations d'influence mutuelle peuvent conduire à rendre inopérante la protection normalement fournie par ces mécanismes (*e.g.* une politique assignant beaucoup de *capabilities* pourrait désactiver une configuration SecComp stricte qui n'autorise que quelques appels système). "*Each kernel security mechanism (including Namespace, Cgroup, SecComp, Capabilities and MAC) restricts kernel permissions from different angles in a fragmented way, while the relationships among them are intricate and complicate. Improper configuration of these security mechanisms might lower their protective capability*" [34, p.425]. Les mécanismes de protection CPU quant à eux (KASLR, SMAP, SMEP), destinés à prévenir les attaques sur le noyau, ont généralement pu être outrepassés depuis l'intérieur d'un *container*.

Concernant plus spécifiquement la fuite d'informations, Gao *et al.* ont adopté une approche systématique pour explorer et identifier les canaux de fuite depuis un *container* qui pourraient exposer des informations sur l'hôte ou les *containers* co-résidents [17, p.1]. Ils ont réalisé leurs tests tout d'abord localement sur des systèmes Linux dotés de configurations par défaut pour Docker et LxC, puis auprès de cinq fournisseurs publics de services *cloud*. Ils ont examiné les différentes possibilités pour exploiter des canaux cachés permettant une communication entre *containers* normalement isolés. Un tel *covert channel* vise l'exploitation discrète de ressources partagées afin de récupérer de l'information sensible et outrepasser les contrôles d'accès établis sur les canaux standards.

Ainsi, Linux propose deux types d'interfaces entre le noyau et l'espace utilisateur : les appels système et les pseudo systèmes de fichiers basés en mémoire (*e.g. procfs*, *sysfs*,...). Ce sont ces derniers qui ont fait l'objet d'une analyse systématique de la manière suivante : "*we design a cross-validation tool to automatically discover these memory-based pseudo files that expose host information to containers. The key idea is to recursively explore all pseudo files under procfs and sysfs in two execution contexts, one running within an unprivileged container and the other running on the host. We align and reorder the files based on their file paths and then conduct pair-wise differential analysis on the contents of the same file between these two contexts. If the system resources accessed from a specific pseudo file has not been namespaced in the Linux kernel, the host and container reach the same piece of kernel data. Otherwise, if properly namespaced, the container can retrieve its own private and customized kernel data*" [17, p.4]. Les différentes ressources ainsi identifiées ont été classées sur base de trois métriques :

— Unicité : le canal permet d'identifier de manière unique une machine hôte, et donc savoir si deux *containers* partagent le même hôte.
— Variation : le canal identifie le changement des données dans le temps, permettant donc d'estimer avec une certaine vraisemblance si deux *containers* résident sur le même hôte (par exemple relever la mémoire libre disponible en même temps).

— Manipulation : le canal peut être manipulé afin d'y "écrire" des données et donc disposer par cet intermédiaire d'un moyen de communication (par exemple détourner */proc/locks* qui présente un aperçu de tous les verrous de l'OS, et non du *container*, pour fabriquer des fichiers symbolisant soit 1 soit 0)

Au départ des informations récoltées et échangées de manière coordonnée via les canaux de fuite d'information mentionnés précédemment, ils ont pu mener plus efficacement une attaque de surconsommation électrique, laquelle provoque l'extinction forcée des serveurs d'un même rack ou alimentés via le même PDU à cause d'une charge de calcul trop soudaine faisant dépasser la consommation électrique maximale sur cette ligne. Pour mener à bien une telle attaque, il faut essentiellement réunir deux conditions, pour lesquelles l'utilisation de canaux cachés de communication s'avère pertinente :

— Premièrement, un maximum de *containers* engagés pour mener l'attaque doivent se trouver sur le même hôte physique ou du moins sur des hôtes partageant la même alimentation électrique. Or, dans une infrastructure IaaS, il n'y a aucune garantie que plusieurs instances créées successivement chez le fournisseur se trouveront hébergées sur le même serveur. L'utilisation des canaux offrant une fuite d'information relative à l'unicité ou la variation laisse savoir si plusieurs *containers* sont co-résidents. Dans la négative, il suffit de détruire l'instance et d'en recréer une autre puis refaire le test. En répétant cette opération, la probabilité augmente pour atteindre l'objectif. Notons que l'on pourrait aussi se limiter à créer toujours plus de *containers*, mais cela occasionnerait des frais de souscription toujours croissants.

— Deuxièmement, une coordination doit avoir lieu entre eux pour exercer un pic de charge de calcul de manière synchronisée. Ce sont ici les canaux par lesquels il est possible de manipuler les données qui sont déterminants. En effet, les différents *containers*, théoriquement isolés entre eux, peuvent se coordonner pour augmenter la charge de calcul (et donc électrique) de manière progressive et discrète, pour finalement lancer l'attaque via une hausse subite synchronisée.

Notons qu'une attaque ayant pour objectif l'épuisement des ressources d'énergie (*energy-draining attack*) peut avoir un impact désastreux sur les systèmes embarqués dépendants de leur batterie (*e.g.* des téléphones portables mais également des satellites) [41].

Catuogno *et al.* proposent une méthodologie pour identifier les services qui dévient de leur usage normal sous des conditions nominales, signe d'une *power attack* potentielle. Leurs tests ont été effectué sur un Raspberry Pi 3B doté d'un OS Raspbian 8. Plutôt que faire tourner le service désiré sur l'OS natif, il est préférable de le faire fonctionner au sein d'un *container* sur lequel il est possible d'imposer des limites dans l'utilisation de ressources (CPU et réseau notamment).

10 Problématisation et perspectives futures

Sur base de cette revue de littérature, trois problèmes de recherche (PR) principaux nous apparaissent :

PR A) Les mécanismes sous-jacents des *containers* Même si la virtualisation légère est déjà largement utilisée, des améliorations aux mécanismes actuels ou des fonctionnalités supplémentaires seraient attendues.

PR B) La sécurité des *containers* Unanimement présentée comme inférieure à la sécurité des machines virtuelles, des améliorations seraient bienvenues tout en conservant les avantages.

PR C) L'évolution et la diversité des solutions de virtualisation légère Docker, Android et Linux semblent monopoliser l'attention principale alors que d'autres solutions et noyaux existent également.

Diverses questions de recherche (QR) ont été formulées dans le cadre de ces trois problématiques (A, B ou C), dont certaines issues des auteurs eux-mêmes (auquel cas la référence est indiquée en fin de question).

QR A.1 Comment intégrer la virtualisation de la téléphonie que peuvent fournir les *containers* avec des numéros de téléphone indépendants [13, p.11] ?

QR A.2 Comment virtualiser les différents senseurs (bluetooth, GPS, NFS,...) [13, p.11] ?

QR A.3 Comment résoudre le problème du *safe device hotplug* pour empêcher les *containers* de créer un nouveau noeud vers un périphérique [54, p.19] ?

QR A.4 L'implémentation des *cgroups* est-elle améliorée en vue d'intégrer toutes les caractéristiques de *rlimits* [54, p.19] ?

QR A.5 Comment implémenter un *device namespace* dans le noyau Linux [54, p.12 et p.18] ?

QR A.6 LxC est-il encore le seul système à supporter tous les modes MACVLAN [54, p.15] ?

QR A.7 Shuttle pourrait-il être implémenté sur des versions récentes de Windows [63] ?

QR A.8 Comment optimiser le temps de démarrage d'un système depuis un périphérique OTG [98, p.622] ?

QR A.9 Comment empêcher les goulots d'étranglement au niveau des réseaux de *containers* [70, p.2] ?

QR A.10 Comment dresser une comparaison objective entre les noyaux (*e.g.* Linux, Unix, Windows,...) pour déterminer les forces et faiblesses respectives de chacun au regard des *containers* ?

QR A.11 Quelle est l'influence des travaux de recherche académiques sur les organisations (*e.g.* Linux Kernel Organization) ou entreprises (*e.g.* Docker) participant à la réalisation effective de technologies de *containers* ?

QR B.1 Comment prévoir un système pour mesurer la qualité de service au sein des *containers* en vue de distinguer une faible performance à cause d'une attaque ou simplement car l'application nécessiterait plus de ressources [10, p.8] ?

QR B.2 Comment améliorer le taux de bande passante de la mémoire comme canal caché pour communiquer de l'information en utilisant davantage de niveaux pour transmettre plusieurs bits à la fois [17, p.10] ?

QR B.3 Comment prendre la micro-architecture du processeur en considération pour construire un modèle d'exploitation de canaux cachés plus raffiné [17, p.13] ?

QR B.4 Comment spécifier davantage et implémenter les mécanismes exacts pour les extensions de confiance depuis le matériel basé sur le TPM aux architectures vTPM proposées [22, p.218] ?

QR B.5 Comment introduire un *security namespace* dans le noyau Linux qui rendrait les *containers* "conscients" des LSM [54, p.18] ?

QR B.6 Comment automatiser la distinction entre composants devant être exécutés en environnement sécurisé de ceux qui ne le doivent pas sur SplitDroid [99, p.92] ?

QR B.7 Comment construire des environnements d'exécution isolés sur SplitDroid, reposant à la fois sur la technologie des *containers* et des services *cloud* (tels un service de stockage) [99, p.92] ?

QR B.8 Comment garantir que l'image enregistrée sur le périphérique OTG ne puisse pas être corrompue ?

QR B.9 Comment élaborer une méthodologie de tests uniforme pour comparer la sécurité des différentes solutions de gestion de *containers* et indépendamment du noyau sur lesquelles elles reposent ?

QR B.10 Quel est le niveau de dépendance entre une solution OSLV et les fonctions de sécurité (*e.g.* SGX, TPM,...) proposées par une micro-architecture matérielle ? Par corollaire, quelle est l'efficacité d'une telle solution lorsqu'elle fonctionne sur une micro-architecture matérielle ne proposant pas ces fonctions ?

QR B.11 Comment automatiser l'identification des mécanismes du noyau effectivement exploités par une solution OSLV et reconstituer ainsi un chemin critique d'exécution des *containers* ?

QR B.12 Quels avantages et inconvénients présentent respectivement chaque LSM dans le cadre de l'exécution de *containers* (*e.g.* SELinux, AppArmor, Smack, Tomoyo,...) ?

QR B.13 Comment réduire les vulnérabilités dans le cadre du tryptique CIA (*Confidentiality*, *Integrity*, *Availability*) sur base des trois sens de menace suivants : du *container* vers l'hôte, de l'hôte vers le *container*, d'un *container* vers un autre *container*, comme le synthétise le tableau 1 ?

Isolation	*Confidentialiy*	*Integrity*	*Availability*
Isolation du *container* vers l'hôte	Le *container* est incapable d'accéder à des ressources de l'hôte que celui-ci n'aurait pas autorisées	Le *container* est incapable d'altérer ou endommager l'hôte	Le *container* est incapable de rendre l'hôte indisponible
Isolation de l'hôte vers le *container*	L'hôte est incapable d'accéder à des ressources du *container* que celui-ci n'aurait pas autorisées	L'hôte est incapable d'altérer ou endommager le *container*	L'hôte est incapable de rendre le *container* indisponible
Isolation d'un *container* vers un autre sur le même hôte	Le *container* est incapable d'accéder à des ressources d'un autre *container* que celui-ci n'aurait pas autorisées	Le *container* est incapable d'altérer ou endommager un autre *container*	Le *container* est incapable de rendre un autre *container* indisponible

TABLE 1 – Matrice de sécurité relative à l'isolation des *containers*

QR C.1 Existe-t-il une solution OSLV pour Mac OS ou iOS [54, p.17] ?

QR C.2 Pourquoi les solutions OSLV tierces à Docker sont-elles peu traitées dans la littérature alors que soutenues par des entreprises représentatives (*e.g.* Canonical pour LxC, Red Hat pour rkt,...) ?

QR C.3 Comment Windows Subsystem for Linux (WSL2) [21] va-t-il influencer les solutions OSLV sur Windows ?

QR C.4 Comment porter des solutions OSLV autres que LxC (*e.g.* Docker) sur Android ?

QR C.5 Quel est l'inventaire exhaustif des solutions OSLV et des noyaux les supportant ?

11 Conclusion

Au terme de cette revue de littérature, nous observons que les mécanismes sous-jacents aux *containers* constituent un sujet suscitant l'intérêt de la communauté scientifique, bien que les travaux de recherche portent principalement sur les plateformes basées sur Linux (Docker et Android en particulier) et Windows 2000/XP dans une moindre mesure.

En matière de sécurité, les avis convergent pour estimer un niveau d'isolation moindre que dans le cadre d'une virtualisation matérielle basée sur un hyperviseur. Mais cela n'empêche pas l'utilisation soutenue des *containers* chez les fournisseurs de service *cloud* pour les avantages que cette technologie comporte en termes de performance et de facilité de déploiement.

Considérant que les solutions populaires actuelles tendent à privilégier l'utilisation d'un noyau Linux *mainstream* (et ne pas appliquer de *patch* à celui-ci), ces solutions voient leur capacité d'isolation réduite aux fonctionnalités rendues possibles par le noyau lui-même. Or l'amélioration de la sécurité globale impliquerait, selon certains auteurs, une refonte non triviale de certains pans du code source Linux.

12 Sigles

BYOD Bring Your Own Device

CCSP Cloud Computing Software Platform

CPU Central Processing Unit

CR Checkpoint and Restart

DAC Discretionary Access Control

DHCP Dynamic Host Configuration Protocol

DLL Dynamic Link Library

EPC Enclave Page Cache

GUI Graphical User Interface

HA High Availability

HAL Hardware Abstraction Layer

HPC High Performance Computing

HTTP HyperText Transfer Protocol

HVV HyperVisor Virtualization

IaaS Infrastructure as a Service

ID IDentifier

I/O Input/Output

IoT Internet of Things

IPC Inter-Process Communication

ISA Instruction Set Architecture

KASLR Kernel Address Space Layout Randomization

KVM Kernel Virtual Machine

LOC Lines Of Code

LSM Linux Security Module

MAC Mandatory Access Control

MitM Man-in-the-Middle

NIS Network Information System

NSA National Security Agency

OCI Open Containers Initiative

OS Operating System

OSI Open Systems Interconnection

OSLV OS-Level Virtualization
OTG On-The-Go
PaaS Platform as a Service
PDU Power Distribution Unit
PKI Public Key Infrastructure
PID Process ID
PIN Personal Identification Number
RAPL Running Average Power Limit
RHEL Red Hat Enterprise Linux
SBC Single Board Computing
SCC Secure Computing Corporation
SCM Service Control Manager
SCONE Secure CONtainer Environment
SDN Software-Defined Networking
SGX Software Guard eXtensions
SMAP Supervisor Mode Access Prevention
SMEP Supervisor Mode Execution Prevention
SMS Short Message Service
SPOF Single Point Of Failure
SSO Single Sign-On
TCB Trusted Computing Base
TCG Trusted Computing Group
TIPC Transparent Inter-Process Communication
TPM Trusted Platform Module
UTS Unix Time Sharing
VE Virtual Environment
VFS Virtual File System
VM Virtual Machine
VMM Virtual Machine Monitor
VNFs Virtual Network Functions
VNI Virtualized Network Interface
VP Virtual Phone
VPN Virtual Private Network
VPS Virtual Private Server
vTPM virtual Trusted Platform Module
XML eXtensible Markup Language

13 Terminologie

On-The-Go Spécification du protocole USB permettant aux périphériques qui en sont équipés de communiquer directement entre eux, sans nécessiter le fait de passer par l'intermédiaire d'un ordinateur personnel [95], *e.g.* une clé USB connectée directement à un *smartphone*.

Principe du moindre privilège En vue d'augmenter la sécurité d'un système, chaque élément de celui-ci ne doit accéder qu'aux ressources strictement nécessaires pour lui permettre d'assurer son fonctionnement légitime. "*Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized.*" [57, p.1282]

Trusted Computing Base Initialement définie comme "*the combination of kernel and trusted processes*" [56, p.13], elle a été étendue dans le "livre orange" comme "*The security-relevant portions of a system*" [30, p.11] "*which contains all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based*" [30, p.66]. Exprimé de manière pragmatique, la TCB vise l'ensemble des composants matériels et logiciels critiques en matière de sécurité, de sorte qu'une compromission de celle-ci pourrait hypothéquer la sécurité du système dans sa globalité.

14 Méthodologie

Une revue systématique de littérature est guidée par une question de recherche. Celle-ci a été formulée de la sorte : "Quels sont les moyens techniques exploités par les systèmes d'exploitation qui permettent l'isolement sécurisé de *containers* ?"

La question a dès lors été opérationnalisée de la manière (technologiquement neutre en regard du moteur de recherche utilisé) suivante (la liste de points étant cumulative) :

— Seuls sont considérés les articles (*i.e.* non les chapitres de monographies)
— Seuls sont considérés les textes rédigés en anglais
— Seules sont considérées les publications postérieures ou égales à 2013
— Le texte [1] doit contenir "operating system level virtualization" OU "os-level virtualization" OU "light virtualization"
— Le texte doit contenir "isolation"
— Le texte doit contenir "container"
— Le texte doit contenir "security"
— Le titre ne doit pas contenir "performance"

Nous avons fait le choix de ne pas orienter la recherche sur des technologies déterminées et avons opté pour des termes génériques usités dans la communauté scientifique par rapport à la thématique envisagée.

Les moteurs de recherche consultés sont les suivants et, pour chacun, figure la requête exacte qui a été exécutée [2] ainsi que la quantité de résultats obtenus de même que les éventuelles particularités du moteur en question [3] :

1. Par l'usage de ce terme général, nous visons n'importe quel champ de métadonnées, voire le corpus complet de la publication.
2. La date ayant été arrêtée au 4 mars 2019.
3. Notons que l'exclusion de termes du titre n'était pas possible de manière automatisée sur tous les moteurs. Cette opération s'est donc déroulée manuellement le cas échéant.

SpringerLink 27 résultats. La catégorie *Computer Science* fait office de filtre parmi les disciplines disponibles, tout comme *Conference Paper* et *Article* dans les types de contenu

> ("operating system level virtualization" OR "os-level virtualization" OR "light virtualization") AND isolation AND container AND security

ScienceDirect 11 résultats. La catégorie *Research Articles* fait office de filtre parmi les différents types disponibles.

> ("operating system level virtualization" OR "os-level virtualization" OR "light virtualization") isolation container security

IEEE Explore 124 résultats.

> ((((((("operating system level virtualization" OR "os-level virtualization" OR "light virtualization")) AND isolation) AND container) AND security) NOT "Document Title" :performance)

ACM 8 résultats.

> (+isolation +container +security) AND content.ftsec :("operating system level virtualization" "os level virtualization" "light virtualization") AND acmdlTitle :(-performance)

arXiv 7 résultats.

> order : -announced_date_first ; size : 50 ; date_range : from 2013-01-01 to 2019-12-31 ; classification : Computer Science (cs) ; include_cross_list : True ; terms : AND all="operating system level virtualization" ; OR all="os-level virtualization" ; OR all="light virtualization" ; AND all=isolation ; AND all=container ; AND all=security ; NOT title=performance

De cette compilation de 177 résultats, nous avons ôté 5 doublons. Le choix entre deux doublons s'est porté sur la publication la plus récente ou bien l'article le plus "complet" (entendu comme volumineux par l'ajout d'une section supplémentaire par exemple) des deux.

Ensuite, en appliquant la méthode recommandée par [45], un tri a été opéré sur base du titre et de l'*abstract* des articles restants. Comme mentionné par les auteurs, cette méthode comporte immanquablement une part de subjectivité. Les thèmes considérés comme non pertinents par rapport à la question de recherche ont été retirés de la liste des documents pris en considération, c'est-à-dire notamment :

- Les comparaisons entre les *containers* et la virtualisation matérielle.
- Les modèles d'architecture proposés dans le cadre d'infrastructure dense de type *cloud*.
- Les thématiques relatives à l'IoT, au *fog computing* ou à l'*edge computing*.
- La virtualisation de réseaux.
- Les tests de performance.
- Les expériences qui faisaient usage de *containers* sans que ceux-ci en constituent le sujet central.
- Les logiciels destinés à gérer le cycle de vie des *containers* et de leurs inter-dépendances, généralement appelés orchestrateurs.

Lorsque le seul résumé des auteurs ne permettait pas de se faire une idée sur la pertinence du document au regard de l'objet de cette revue de littérature (nous visons les références identifiées comme "*doubtful*" selon [45]), un filtre a été opéré à la lecture de l'introduction et de la conclusion de chaque article concerné. Ainsi, au départ des 177 articles obtenus automatiquement sur base des critères de recherches énumérés et des moteurs documentaires mentionnés, après retrait des doublons et en passant par l'examen qui vient d'être décrit, nous obtenons un *corpus* de 40 références (6 chez SpringerLink, 2 chez ScienceDirect, 22 chez IEEE Explore, 5 chez ACM, 5 chez arXiv). Notons qu'il ne nous a pas été possible d'accéder au contenu de l'une des références ayant traversé tous les filtres décrits, à savoir [4].

15 Bibliographie

Les références comportant un astérisque (*) en fin de citation constituent des ajouts par rapport aux éléments bibliographiques obtenus via la méthodologie présentée sous le titre 14. Leur mention se justifie pour l'une des raisons suivantes :

- La référence comporte des informations pertinentes sur une thématique qui n'est pas spécifiquement visée par la revue de littérature elle-même (typiquement la méthodologie utilisée pour constituer celle-ci).
- La référence apparaît comme une ressource documentaire incontournable au regard de la thématique envisagée, mais est simplement introuvable par le biais des moteurs de recherche décrits sous le titre 14 visant spécifiquement la littérature scientifique (par exemple un *website* essentiel par rapport aux technologies développées).
- La référence est apparue dans l'un ou plusieurs des articles obtenus via la méthodologie développée sous le titre 14 et il nous semblait pertinent de l'ajouter (par exemple car il s'agit d'un article originel exposant une "nouvelle" technologie) bien qu'elle ne soit pas ressortie de notre requête automatisée (par exemple, car elle est antérieure à la période de publication qui avait été prise en considération).

Notons enfin que les éléments temporels (mois et année) des références ciblant un *website* indiquent le moment lors duquel celui-ci a été consulté dans le cadre de cette revue de littérature.

Références

[1] P. Adkins. Talos vulnerability report : Alpine linux docker image root user hard-coded credential vulnerability. `https://talosintelligence.com/vulnerability_reports/TALOS-2019-0782`, May 2019. *.

[2] J. Andrus, C. Dall, A. Hof, O. Laadan, and J. Nieh. Cells : a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187, October 2011. *.

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. Scone : Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 689–703, November 2016.

[4] B. Arnold, S. Baset, P. Dettori, M. Kalantar, I. Mohomed, S. Nadgowda, M. Sabath, S. Seelam, M. Steinder, M. Spreitzer, and A. Youssef. Building the ibm containers cloud service. *IBM Journal of Research and Development*, 60(2-3) :9 :1–9 :12, March 2016. doi : 10.1147/JRD.2016.2516943.

[5] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi. Dockerpolicymodules : Mandatory access control for docker containers. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 749–750, September 2015. doi : 10.1109/CNS.2015.7346917.

[6] M. Bauer. Paranoid penguin - an introduction to novell apparmor. `https://www.linuxjournal.com/article/9036`, July 2006. *.

[7] E. Biederman. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112, July 2006. *.

[8] P. Brereton, B. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4) :571–583, April 2007. doi : 10.1016/j.jss.2006.07.009. *.

[9] L. Catuogno and C. Galdi. On the evaluation of security properties of containerized systems. In *2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS)*, pages 69–76, December 2016. doi : 10.1109/IUCC-CSS.2016.018.

[10] L. Catuogno, C. Galdi, and N. Pasquino. An effective methodology for measuring software resource usage. *IEEE Transactions on Instrumentation and Measurement*, 67(10) :2487–2494, Oct 2018. doi : 10.1109/TIM.2018.2815431.

[11] P. Chanezon. Docker for mac and windows beta : the simplest way to use docker on your laptop. `https://blog.docker.com/2016/03/docker-for-mac-windows-beta`, May 2019. *.

[12] C. Chen, Z. Zhang, and X. Xie. Container cloud resource allocation based on combinatorial double auction. In *Proceedings of the 3rd International Conference on Intelligent Information Processing*, pages 146–151, May 2018. doi : 10.1145/3232116.3232140.

[13] W. Chen, L. Xu, G. Li, and Y. Xiang. A lightweight virtualization solution for android devices. *IEEE Transactions on Computers*, 64(10) :2741–2751, October 2015. doi : 10.1109/TC.2015.2389791.

[14] V. Costan and S. Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. *.

[15] R. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5) :483–490, September 1981. *.

[16] H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. Rapl : Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, August 2010. doi : 10.1145/1840845.1840883. *.

[17] X. Gao, B. Steenkamer, Z. Gu, M. Kayaalp, M. Pendarakis, and H. Wang. A study on the security implications of information leakages in container clouds. *IEEE Transactions on Dependable and Secure Computing*, pages 1–17, November 2018. doi : 10.1109/TDSC.2018.2879605.

[18] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, A. López-García, and P. Orviz. Enabling rootless linux containers in multi-user environments : The udocker tool. *Computer Physics Communications*, 232 :84–97, November 2018. doi : https://doi.org/10.1016/j.cpc.2018.05.021.

[19] J. Gray and D. Siewiorek. High-availability computer systems. *Computer*, 24(9) :39–48, September 1991. doi : 10.1109/2.84898. *.

[20] S. Hallyn and A. Morgan. Linux capabilities : making them work. In *Proceedings of the Linux Symposium*, volume 1, pages 163–172, July 2008. *.

[21] J. Hammons. Shipping a linux kernel with windows. `https://devblogs.microsoft.com/commandline/shipping-a-linux-kernel-with-windows`, May 2019. *.

[22] S. Hosseinzadeh, S. Laurén, and V. Leppänen. Security in container-based virtualization through vtpm. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pages 214–219, December 2016.

[23] M. Huber, J. Horsch, M. Velten, M. Weiss, and S. Wessel. A secure architecture for operating system-level virtualization on mobile devices. In *Information Security and Cryptology*, pages 430–450, May 2016. doi : 10.1007/978-3-319-38898-4_25.

[24] P.-H. Kamp and R. Watson. Jails : Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, pages 116–131, May 2000. *.

[25] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai. Multilanes : Providing virtualized storage for os-level virtualization on many cores. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 317–329, February 2014.

[26] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai. Multilanes. `https://github.com/kangjunbin/MultiLanes`, April 2019. *.

[27] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering (version 2.3). Technical report, Keele University, July 2007. *.

[28] K. Kolyshkin. Virtualization in linux. *White paper, OpenVZ*, 2006. *.

[29] S. Kywe, C. Landis, Y. Pei, J. Satterfield, Y. Tian, and P. Tague. Privatedroid : Private browsing mode for android. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 27–36, September 2014. doi : 10.1109/TrustCom.2014.8.

[30] D. Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986. *.

[31] M. Lee, M. Lee, I. Kim, and Y. Eom. User isolation in multi-user multi-touch devices using os-level virtualization. In *Cloud Computing*, pages 30–38, May 2016. doi : 10.1007/978-3-319-38904-2_4.

[32] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li. Speaker : Split-phase execution of application containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 230–251, June 2017. doi : 10.1007/978-3-319-60876-1_11.

[33] S. Liao, X. Yang, W. Guo, H. Sun, Z. Jiang, and X. Zhao. Gemini : A lightweight virtualization architecture for protecting privacy and security of smartphone. In *2016 13th International Conference on Embedded Software and Systems (ICESS)*, pages 186–191, August 2016. doi : 10.1109/ICESS.2016.27.

[34] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou. A measurement study on linux container security : Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 418–429, December 2018. doi : 10.1145/3274694.3274720.

[35] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX track : USENIX Annual Technical Conference*, June 2001. *.

[36] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris. Docker-sec : A fully automated container security enhancement mechanism. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1561–1564, July 2018. doi : 10.1109/ICDCS.2018.00169.

[37] A. Madhavapeddy. Improving docker with unikernels : Introducing hyperkit, vpnkit and datakit. `https://blog.docker.com/2016/05/docker-unikernels-open-source`, May 2019. *.

[38] A. Madhavapeddy and R. Mortier. Advanced docker developer workflows on macos x and windows. `https://www.slideshare.net/AnilMadhavapeddy/advanced-docker-developer-workflows-on-macos-x-and-windows`, May 2019. *.

[39] A. Manu, J. Patel, S. Akhtar, V. Agrawal, and K. Murthy. A study, analysis and deep dive on cloud paas security in terms of docker container security. In *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, pages 1–13, March 2016. doi : 10.1109/ICCPCT.2016.7530284.

[40] A. Manu, J. Patel, S. Akhtar, V. Agrawal, and K. Murthy. Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. In *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, pages 1–14, March 2016. doi : 10.1109/ICCPCT.2016.7530217.

[41] T. Martin, M. Hsiao, D. Ha, and J. Krishnaswami. Denial-of-service attacks on battery-powered mobile computers. In *Second IEEE Annual Conference on Pervasive Computing and Communications, 2004.*, pages 309–318, March 2004. *.

[42] P. Mendki. Docker container based analytics at iot edge. video analytics usecase. In *2018 3rd International Conference On Internet of Things : Smart Innovation and Usages (IoT-SIU)*, pages 1–4. Birla Institute of Applied Sciences Bhimtal, February 2018. doi : 10.1109/IoT-SIU.2018.8519852. *.

[43] A. Mirkin, A. Kuznetsov, and K. Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, July 2008. *.

[44] A. Natawiguna and M. Liem. Virtualization methods for securing online exam. In *2016 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–7, October 2016. doi : 10.1109/ICODSE.2016.7936145.

[45] F. Neiva and R. Silva. Systematic literature review in computer science - a practical guide. Technical report, Federal University of Juiz de Fora, November 2016. *.

[46] O. Oluwatimi, D. Midi, and E. Bertino. Overview of mobile containerization approaches and open research directions. *IEEE Security Privacy*, 15(1) :22–31, January 2017. doi : 10.1109/MSP.2017.12.

[47] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap : A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002. *.

[48] S. Park, J. Kim, and D. Lee. Securedom : secure mobile-sensitive information protection with domain separation. *The Journal of Supercomputing*, 72(7) :2682–2702, July 2016. doi : 10.1007/s11227-015-1578-6.

[49] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in plan 9. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop : Models and paradigms for distributed systems structuring*, pages 1–5. ACM, September 1992. *.

[50] H. Potzl. Linux-vserver technology. *LinuxTAG 2004*, June 2004. *.

[51] O. Pozdniakova and D. MaZeika. A cloud software isolation and cross-platform portability methods. In *2017 Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–6, April 2017. doi : 10.1109/eStream.2017.7950315.

[52] D. Price and A. Tucker. Solaris zones : Operating system support for consolidating commercial workloads. In *LISA*, volume 4, pages 241–254, July 2004. *.

[53] B. Reed, N. MacDonald, P. Firstbrook, S. Olyaei, and P. Bhajanka. Top 10 security projects for 2019. `https://www.gartner.com/doc/3900996/top--security-projects-`, April 2019. *.

[54] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan. Security of os-level virtualization technologies : Technical report. *Computer Research Repository (arXiv)*, July 2014.

[55] C. Rotter, L. Farkas, G. Nyíri, G. Csatári, L. Jánosi, and R. Springer. Using linux containers in telecom applications. In *19th International ICIN Conference*, pages 234–241, March 2016. *.

[56] J. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 12–21, December 1981. doi : 10.1145/800216.806586. *.

[57] J. Saltzer and M. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975. doi : 10.1109/PROC.1975.9939. *.

[58] N. Santos, K. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Usenix HotCloud*, June 2009. *.

[59] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension : Using sgx to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24, June 2017. doi : 10.1007/978-3-319-60876-1_1.

[60] M. Schwarz, S. Weiser, and D. Gruss. Practical enclave malware with intel sgx. *Computer Research Repository (arXiv)*, February 2019. *.

[61] V. Sekar, V. Patil, M. Giusti, A. Bhide, and A. Gupta. Aws ec2 vs. joyent's triton : A comparison of docker container-hosting platforms. In *Proceedings of the 8th Workshop on Scientific Cloud Computing*, pages 33–36, June 2017. doi : 10.1145/3086567.3086572.

[62] M. Shah, M. Kamran, H. Khan, and Q. Javaid. Vdroid : A lightweight virtualization architecture for smartphones. In *2016 Future Technologies Conference (FTC)*, pages 1290–1296, December 2016. doi : 10.1109/FTC.2016.7821766.

[63] Z. Shan, X. Wang, and T. Chiueh. Shuttle : Facilitating inter-application interactions for os-level virtualization. *IEEE Transactions on Computers*, 63(5) :1220–1233, May 2014. doi : 10.1109/TC.2012.297.

[64] Z. Shan, T. Chiueh, and X. Wang. Duplication of windows services. *Computer Research Repository (arXiv)*, September 2016.

[65] Z. Shan, T. Chiueh, and X. Wang. Virtualizing system and ordinary services in windows-based os-level virtual machines. *Computer Research Repository (arXiv)*, September 2016.

[66] Z. Shan, Y. Yu, and T. Chiueh. Confining windows inter-process communications for os-level virtual machine. *Computer Research Repository (arXiv)*, September 2016.

[67] S. Singh and N. Singh. Containers amp ;amp ; docker : Emerging roles amp ;amp ; future of cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pages 804–807, July 2016. doi : 10.1109/ICATCCT.2016.7912109.

[68] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, December 2001. *.

[69] V. Sochat. The open container initiative (oci) part 1 : The problems that containers solve. `https://youtu.be/cJp86kGOAQg`, April 2019. *.

[70] K. Suo, Y. Zhao, W. Chen, and J. Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 189–197, April 2018. doi : 10.1109/INFOCOM.2018.8485865.

[71] M. Verma and M. Dhawan. Towards a more reliable and available docker-based container cloud. *Computer Research Repository (arXiv)*, August 2017.

[72] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li. Mining sandboxes for linux containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, March 2017. doi : 10.1109/ICST.2017.16.

[73] S. Wessel, M. Huber, F. Stumpf, and C. Eckert. Improving mobile device security with operating system-level virtualization. *Computers and Security*, 52 :207–220, July 2015. doi : https://doi.org/10.1016/j.cose.2015.02.005.

[74] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security module framework. In *Proceedings of the Ottawa Linux Symposium*, pages 604–617, June 2002. *.

[75] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules : General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, August 2002. *.

[76] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. Airbag : Boosting smartphone resistance to malware infection. In *Network and Distributed System Security Symposium*, February 2014. *.

[77] X. Alpine linux. `https://www.alpinelinux.org`, May 2019. *.

[78] X. Cellrox. `http://www.cellrox.com`, April 2019. *.

[79] X. Install docker desktop for mac. `https://docs.docker.com/docker-for-mac/install`, May 2019. *.

[80] X. Install docker desktop for windows. `https://docs.docker.com/docker-for-windows/install`, May 2019. *.

[81] X. Exploit-db. `https://www.exploit-db.com`, April 2019. *.

[82] X. Flask : Flux advanced security kernel. `https://www.cs.utah.edu/flux/fluke/html/flask.html`, April 2019. *.

[83] X. Illumos. `https://illumos.org`, May 2019. *.

[84] X. Intel software guard extensions. `https://www.intel.fr/content/www/fr/fr/architecture-and-technology/software-guard-extensions.html`, April 2019. *.

[85] X. Linux containers. `https://linuxcontainers.org`, April 2019. *.

[86] X. Customer guidance for cve-2019-0708. `https://support.microsoft.com/en-us/help/4500705/customer-guidance-for-cve-2019-0708`, May 2019. *.

[87] X. capabilities(7) linux manual page. `http://man7.org/linux/man-pages/man7/capabilities.7.html`, April 2019. *.

[88] X. chroot(2) linux manual page. http://man7.org/linux/man-pages/man2/chroot.2.html, April 2019. *.

[89] X. Namespaces(7) linux manual page. http://man7.org/linux/man-pages/man7/namespaces.7.html, April 2019. *.

[90] X. Open containers initiative. https://www.opencontainers.org, April 2019. *.

[91] X. rkt. https://github.com/rkt/rkt, April 2019. *.

[92] X. Seccomp bpf (secure computing with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, April 2019. *.

[93] X. Joyent smartos. https://www.joyent.com/smartos, May 2019. *.

[94] X. Trusted computing group. https://trustedcomputinggroup.org, April 2019. *.

[95] X. Usb on the go and embedded host. https://www.usb.org/usb-on-the-go, April 2019. *.

[96] X. Linux-vserver. http://linux-vserver.org, April 2019. *.

[97] X. udocker. https://github.com/indigo-dc/udocker, April 2019. *.

[98] Y. Xue, X. Zhang, X. Yu, Y. Zhang, Y. Tan, and Y. Li. Isolating host environment by booting android from otg devices. *Chinese Journal of Electronics*, 27(3) :617–624, August 2018. doi : 10.1049/cje.2018.03.017.

[99] L. Yan, Y. Guo, and X. Chen. Splitdroid : Isolated execution of sensitive components for mobile applications. In *Security and Privacy in Communication Networks*, pages 78–96, 2015. doi : 10.1007/978-3-319-28865-9_5.

[100] X. Yang and C. Sun. Research and implementation of multiple android systems based on the container technique. *Journal of Chinese Computer System*, 7 :1422–1427, 2016. *.

[101] Y. Yu, F. Guo, S. Nanda, L. Lam, and T. Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 24–34, June 2006. *.

[102] M. Zhang, D. Marino, and P. Efstathopoulos. Harbormaster : Policy enforcement for containers. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 355–362, November 2015. doi : 10.1109/CloudCom.2015.96.