MASTER 2 SRIV

Université Claude Bernard Lyon 1

Analyse de la performance de la technologie de sécurité Intel SGX

Auteur :

Maysa Abou Jamra

Lieu de stage :

iExec
Blockchain
Tech
22 Janvier - 24 Août 2018

Encadrement :

Maître de stage : **Lei Zhang**

Tuteur pédagogique : **Thomas Begin**

Année universitaire 2017-2018

Sommaire

Table des figures

Remerciements

Je tiens tout d'abord à remercier Jésus qui m'a donné la force de garder confiance et courage malgré les épreuves difficiles que j'ai vécu surtout cette année.

En second lieu, j'adresse mes remerciements à mon professeur, M. Yves Caniou de l'Université Claude Bernard Lyon 1 qui m'a beaucoup aidé dans ma recherche de stage. J'apprécie énormément l'effort qu'il a fait pour nous permettre moi et ma copine libanaise de s'intégrer rapidement dans la classe sachant que c'était notre première année à l'étranger. Je vais jamais oublier aussi l'intérêt qu'il a porté à mon choix du sujet de stage et ses conseils qui m'ont permis de postuler dans iExec.

Je remercie également M. Lei Zhang, qui a supervisé mon stage, pour m'apporter le soutien moral nécessaire et surtout pour l'autonomie qu'il m'a offert dans mon travail. Merci également à Mademoiselle Delphine, la secrétaire aimable de l'entreprise, qui était toujours à l'écoute. J'ai beaucoup apprécié aussi l'accueil chaleureux de toute l'équipe de iExec et je les remercie pour l'ambiance agréable de travail qu'ils ont créé.

Je voudrais aussi exprimer ma reconnaissance envers ma famille et surtout ma mère qui a fait tout type de sacrifice pour me rendre heureuse. Un grand merci à mes amis Zahiya, Zeina, Youssef et Zied qui ont su m'épauler tout au long de cette formation et qui par leurs encouragements j'ai pu surmonter tous les obstacles.

Enfin, mes plus grands remerciements vont à ceux qui m'ont blessé, à ceux qui ont quitté ma vie quand j'avais le plus besoin de leur présence car c'est grâce à eux je sais plus ce que je veux.

Résumé / Abstract

Les technologies de sécurité utilisées pour protéger les applications Docker n'évitent pas les risques des attaquants profitant des compromis du OS pour inspecter les données de ces applications ou du risque provenant de ceux ayant un accès privilégié sur la machine où s'exécutent ces données. Dans un environnement Cloud, les clients ont besoin de protéger leurs données même du fournisseur du Cloud qui possède un accès privilégié sur les machines et peut inspecter les données des utilisateurs. Intel SGX, une technologie de sécurité développée en 2015, utilise des mécanismes matérielles pour protéger les données des clients utilisant des applications qui s'exécutent sur des machines distantes (Cloud) d'être inspectées même par les possesseurs de ces machines. À cause du processus compliqué qu'elle utilise pour chiffrer les secrets et de quelques limitations, elle peut affecter la performance des applications. Dans le but de comprendre la performance de Intel SGX et d'étudier des limitations, le sujet de stage est focalisé sur les tests de mesure de performance avec différents métriques pour analyser à quel point la sécurité assurée par SGX peut dégrader son efficacité. Dans ce rapport, je justifie dans un premier lieu l'importance de la sécurité pour les applications Docker utilisées dans le Cloud et je décris le fonctionnement de SGX, son intérêt dans ce domaine et ses limitations. Ensuite, j'explique les spécificités du conteneur Docker SCONE qui est sécurisé par SGX et l'amélioration qu'il a effectué par rapport au fonctionnement basique de SGX. Plus tard, je justifie les métriques de performance étudiées et j'expose les outils de mesure dont je me suis servie, les tests que j'ai effectuées et les résultats obtenus de ces essais.

Security technologies used to protect Docker applications do not avoid the risk of attackers taking advantage of OS compromises to inspect data from these applications or the risk from those with privileged access to the machine where the data is running. In a cloud environment, customers need to protect their data even from the cloud provider who has privileged access to machines and can inspect user data. Intel SGX, a security technology developed in 2015, uses hardware mechanisms to protect data from clients using applications that run on remote machines (Cloud) to be inspected even by the owners of these machines. Because of the complicated process it uses to encrypt secrets and some limitations, it can affect application performance. In order to understand the performance of Intel SGX and study its limitations, the internship topic focuses on performance measurement tests with different metrics to analyze how the security involved in SGX can degrade its effectiveness. In this report, I first justify the importance of security for Docker applications used in the Cloud and describe the features of SGX, its interest in this area and its limitations. Then, I explain the specificities of the Docker SCONE container that is secured by SGX and the improvement it has made compared to the basic operating mode of SGX. Later, I justify the performance metrics studied and I expose the measurement tools that I used, the tests I performed and the results obtained from these tests.

Introduction

Dans le cadre de mon Master 2 intitulé “Systèmes, Réseaux Et Infrastructures Virtuelles SRIV" à l'Université Claude Bernard Lyon 1, j'ai souhaité réaliser mon stage de fin d'études dans le domaine de la sécurité informatique. D'ailleurs, le grand intérêt que montre la blockchain de nos jours m'a motivé de candidater à la société iExec Blockchain Tech pour effectuer mon stage avec leur équipe.

L'entreprise française iExec, créée très récemment en 2016, s'est fait connaître avec succès pour son sujet innovant et ses services fiables et performants. Elle se situe à Lyon, près de la part-Dieu. Le but principal de iExec est de décentraliser le Cloud et de permettre le traitement des données à proximité de leur lieu de stockage. Cette entreprise permet, à ceux qui proposent de la ressource de calcul, d'échanger directement avec les développeurs des applications décentralisées en payant par la monnaie RLC qui est la crypto-monnaie de iExec. Ces applications sont basées sur la *blockchain* Ethereum.

J'ai effectué mon stage à iExec du 22 Janvier 2017 au 24 Août 2018 (5 mois) sous la surveillance de M. Lei ZHANG, Ingénieur chercheur en sécurité et cryptographie, qui m'a soutenu et encouragé tout au long du stage. Le but du stage était d'analyser la performance de Intel SGX qui est une technologie de sécurité développée par Intel en 2015. Ce stage a consisté essentiellement à comparer des outils de mesure Docker, choisir des métriques de mesure de performance avec la technologie Intel SGX, ainsi qu'à analyser des résultats des tests effectués sur les applications natives et celles utilisant SGX. Ces résultats peuvent aider iExec à choisir pour quel type d'applications décentralisées de iExec SGX est rentable.

Ce stage a été l'opportunité pour moi de comprendre l'importance de la notion de performance dans les technologies de sécurité. Au-delà d'enrichir mes connaissances dans le domaine informatique (docker, sécurité, etc.), ce stage m'a permis d'acquérir des compétences relationnelles et de s'adapter au travail en autonomie.

Dans ce rapport, je décrirai l'entreprise iExec et son secteur en insistant sur l'enjeu de ce projet. Puis je présenterai mes missions lors de ce stage et ce qu'il m'a apporté.

Chapitre 1

Environnement de stage

Dans ce chapitre, je vais définir d'abord iExec et son secteur d'activité en expliquant brièvement la blockchain, le rôle de la blockchain Ethereum, ainsi que le but d'iExec qui utilise cette dernière. Ensuite je vais parler de la cryptomonnaie utilisée par iExec et les dApps. Je finis le chapitre par l'explication de ma mission dans cette entreprise.

1.1 iExec et son objectif

iExec Blockchain Tech est une start-up née en 2016, son siège social est à Lyon en France, avec une filiale à Hong Kong [1]. Les fondateurs du projet de l'entreprise sont Gilles Fedak et Haiwwu He.

La blockchain est une technologie qui devient de plus en plus utilisée dans plusieurs secteurs. La blockchain, ou chaîne de blocs, est un registre pour le stockage et l'échange de tout type d'informations ou de valeurs (monnaie électronique, votes, bons de fidélité, etc.). Elle est constituée de blocs contenant l'historique de tous les échanges effectués entre ses utilisateurs depuis sa création. C'est une base de donnée distribuée et sécurisée fonctionnant sans organe de contrôle central. Afin de valider les échanges effectués sur la blockchain, la capacité de cette technologie d'opérer dans un cadre décentralisé nécessite un accord connu sous le nom "consensus" entre un groupe de ses nœuds. Par conséquent, un nœud qu'on appelle "mineur" est sélectionné dans chaque intervalle de temps (variant selon la blockchain utilisée) pour valider les échanges récentes et ajouter un nouveau bloc à la chaîne de blocs.

Les crytomonnaies ou monnaies virtuelles reposent sur la technologie de la blockchain et chacune a son algorithme de consensus [2, 3].

La blockchain Ethereum est une blockchain conçue pour l'exécution des applications décentralisées [4]. Elle permet aux programmeurs d'écrire des contrats intelligents qui seront exécutés sur la blockchain. Un contrat intelligent permet l'exécution automatique de contrats dont les conditions ont été définies dans

l'application. Lors de la réalisation des clauses, le code décide l'exécution du contrat sans besoin d'intervention humaine.

Les blockchains et donc aussi Ethereum offrent des capacités de calcul très limitées pour les applications décentralisées tels que quelques kB de stockage, des machines virtuelles faibles en puissance, etc. iExec est un projet franco-chinois qui a pour but de donner aux applications décentralisées de la blockchain Ethereum un accès à des ressources off-chain (du Cloud) pour remplir leurs besoins. Ce Cloud est décentralisé puisque les serveurs utilisés ne sont pas les serveurs des data centers et c'est pour plusieurs raisons telles que l'énorme consommation de l'énergie par ces data centers et le coût d'utilisation de ces data centers qui est élevé.

L'entreprise iExec collabore avec Stimergy [5] et Qarnot Computing [6] qui sont deux entreprises françaises pour répartir les serveurs dans la ville. Ces serveurs peuvent être, par exemple, des chauffages à proximité des utilisateurs. De cette façon, les données des applications peuvent être traitées rapidement et à moindre coût.

1.2 Monnaie d'iExec

La monnaie utilisée par iExec est le RLC qui veut dire "Runs on Lots of Computers". Le coût de ce type de monnaie change beaucoup sur le marché global [7].

Cette entreprise est aussi une place de marché c'est-à-dire elle permet aux fournisseurs de ressources de calcul et aux développeurs des applications de communiquer entre eux. Les développeurs paient en RLC à ceux qui mettent en disposition les ressources de calcul de leurs serveurs et iExec ne touche pas de commissions lors de ces transactions.

1.3 Les dApps

Les applications décentralisées connues sous le nom de "dApps" fonctionnent sans besoin d'un tiers de confiance. Les dApps, normalement open source, sont révolutionnaires car elles permettent d'établir des transactions pair à pair entre client et fournisseur, basées sur la confiance et la transparence.

Les dApps de iExec se trouvent dans le dapp Store [8]. Les utilisateurs peuvent se servir de ces dApps, tandis que les développeurs peuvent soumettre leurs propres applications et gagner de l'argent s'ils souhaitent les monétiser.

1.4 Équipe

L'équipe de iExec est formée des développeurs d'affaires, des *blockchain hackers*, des spécialistes en *marketing*, ainsi que des stagiaires et des doctorants. L'équipe de iExec cette année était formée de 15 membres fixes et de 7 stagiaires.

1.5 Mes missions dans iExec

Le sujet de stage que j'ai effectué dans iExec était l'analyse de la performance de Intel SGX. Les missions qui m'étaient confiées étaient les suivantes :

1. La compréhension du fonctionnement de SGX et l'analyse de ses points de faiblesse qui affectent sa performance.
2. L'étude du conteneur SCONE qui utilise SGX et dégager ses limitations.
3. Les tests de mesure de performance sur des conteneurs Docker natifs et des conteneurs construis avec SCONE.

Les membres de iExec ont été motivé d'utiliser SGX dans leur plate-forme pour sécuriser les dApps, mais avant de prendre cette décision il fallait tester la performance de cette technologie. Ces tests réalisés aident iExec à choisir pour quel type d'applications SGX peut être rentable.

Chapitre 2

Technologies utilisées et problématique du stage

Dans cette partie, je vais expliquer d'abord les technologies nécessaires que j'ai appris dans mon stage afin de comprendre le contexte du sujet. Dans un second lieu, je vais parler des objectifs et de l'intérêt de mon travail pour l'entreprise.

2.1 Besoin de sécurité pour les dApps de iExec

La sécurité des applications centralisées est assurée par des mécanismes appliqués sur les serveurs centrales qui les gèrent (utilisation des *firewalls* et chiffrement). Les applications de iExec fonctionnent sur la blockchain Ethereum. Cette dernière a des intérêts qu'on ne peut pas dénier mais elle offre des ressources de calcul limitées aux applications tels qu'une capacité de stockage de quelques KB seulement, les machines virtuelles ne sont pas assez efficaces et une latence remarquable d'exécution. Certaines applications demandent plus de capacité de calcul d'où l'idée de iExec de donner à ces applications tournant sur la blockchain l'accès à des ressources off-chain (du Cloud décentralisé). Ces applications décentralisées de iExec ne sont pas gérées par un organe central de contrôle. Par conséquent, on ne peut pas faire confiance aux nœuds distribués de ce Cloud où ces applications qui peuvent avoir des données confidentielles et qui sont exécutées sur ces machines peuvent être modifiées, vendues ou utilisées pour des fins illégales. C'est pour ces raisons que c'est indispensable pour iExec de trouver une solution sécurisante afin de protéger l'exécution des dApps et leurs données.

2.2 Intel SGX : une technologie sécurisante

2.2.1 Introduction

La technologie Intel SGX (*Intel Software Guard Extensions*) est une technologie de sécurité développée par Intel en 2015 à partir de l'architecture Skylake [9,

10]. L'idée d'Intel est que même si les applications qui traitent des données sensibles sont conçues avec soin et validées pour être difficiles à attaquer, un compromis du système d'exploitation de la machine où ils tournent donne à l'attaquant un accès complet à toutes les données de ces applications [11]. Pour cela, le concepteur d'Intel SGX a proposé un nouveau modèle de programmation, qui divise le code de l'application en deux parties de façon à mettre le code et les données confidentielles dans la partie protégée et le reste de l'application dans une partie non fiable.

SGX met à disposition un ensemble de nouvelles instructions du CPU d'Intel qui permettent d'allouer des zones privées de la mémoire, appelées enclaves, dans laquelle le développeur d'application met le code et les données secrètes. Le mécanisme de sécurité utilisé par cette technologie vise à protéger l'enclave spécifique d'une application d'être accessible par le système d'exploitation, l'hyperviseur et les autres applications tournant sur la même machine (voir FIGURE2.1). Une autre fonctionnalité de SGX est d'assurer le calcul sécurisé sur une machine distante. Cette dernière idée était évoquée dans un livre sous le nom de "*Orange Book*" publié en 1983 [12] mais n'était pas implémentée qu'avec SGX. Afin de garantir cette possibilité, Intel fournit le *Quoting Enclave* qui est une enclave différente de l'enclave créée par le développeur de l'application ; c'est une partie de la plate-forme de Intel SGX.

2.2.2 Sécurité dans Intel SGX

SGX est principalement implémentée dans le micro-code du processeur. La sécurité d'Intel SGX repose sur la clé du CPU qui est fusionnée à l'intérieur du processeur pendant la production et que personne n'est capable d'inspecter y compris le fabricant Intel. Cette clé est utilisée pour protéger les applications s'exécutant dans une enclave. Cette zone mémoire (l'enclave) est chiffrée avec des clés dérivées de la clé du CPU par une unité de matériel à l'intérieur du processeur appelé *Memory Encryption Engine* (MEE) qui protège contre les attaques physiques sur la mémoire principale [13]. Le MEE déchiffre et chiffre de manière transparente les lectures et les écritures sur l'enclave ; cela garantit que les données ne sont conservées en texte clair que lorsqu'ils résident dans le cache, à l'intérieur du CPU. Par conséquent, SGX protège la confidentialité et l'intégrité du processus de certaines formes d'attaques *hardware* et d'autres processus sur le même hôte, y compris les processus privilégiés comme les systèmes d'exploitation.

Une enclave peut lire et écrire en dehors de sa région mais aucun autre processus ne peut accéder à la mémoire de l'enclave. Ainsi l'exécution isolée dans SGX protège les données et le code sensible pour s'exécuter correctement, en toute confidentialité et intégrité. Une des restrictions de SGX est que c'est pas possible d'effectuer des appels systèmes dans l'enclave puisque avec SGX on suppose que le OS n'est pas fiable. À chaque fois que l'application veut effectuer un appel système, il faut que le *thread* de l'enclave copie les arguments de la mémoire concernés et sort de l'enclave avant qu'un appel système s'exécute. Ce mécanisme affecte la performance à cause de ces transitions de *threads* qui im-

pliquent la sauvegarde et la restauration de l'état de l'enclave avec chaque transition.

La technologie SGX ne fournit pas des mécanismes de protection systématiques contre les attaques par canaux auxiliaires [14] ; elle s'appuie sur le développeur de l'application. Elle ne protège pas également des attaques qui peuvent décompresser le CPU tel que l'attaque par injection de fautes.

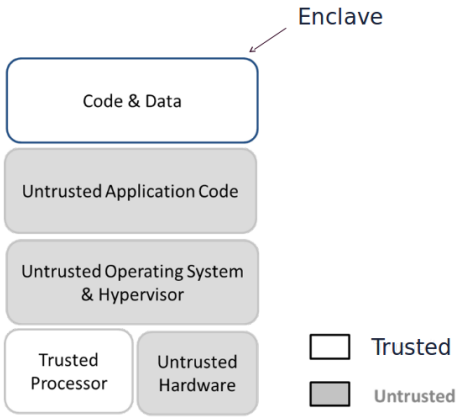

FIGURE 2.1 – Les parties fiables et non fiables dans l'architecture Intel SGX

2.2.3 *Secret Provisioning*

SGX permet à un système distant de vérifier le programme fonctionnant dans une enclave pour lui communiquer les données après en toute sécurité. Quand une enclave est créée, le CPU produit un hachage de son état initial connu sous le nom de *MRENCLAVE measurement*. Quand l'agent distant demande de stocker ses données dans cette enclave, le *Quoting Enclave* qui est un logiciel fournit par Intel, demande un rapport qui inclut le hash *MRENCLAVE* et des données supplémentaires fournies par le processus.

Le rapport est signé numériquement à l'aide d'une clé du *hardware* pour produire une preuve que le programme mesuré fonctionne dans une enclave protégée par SGX. Cette preuve, connue sous le nom de "*Quote*", est une attestation qui peut être vérifiée par un système distant. SGX signe le *Quote* en utilisant une signature de groupe appelée EPID (Enchanced Privacy ID) [15]. Elle est anonyme d'une façon qu'aucun ne peut savoir la clé de signature même son émetteur. Quand l'utilisateur reçoit ce *Quote* signé, il demande une vérification d'un serveur de Intel (IAS). Après cette vérification, l'entité distante peut fournir des secrets à cette enclave SGX ce qu'on appelle "*Secret Provisioning*".

Généralement, l'approvisionnement secret (*Secret Provisioning*) est effectué via un canal très sécurisé. Ce canal est établi entre l'entité distante et l'enclave, et la communication entre eux est chiffrée par la clé de l'enclave qui est générée

de manière aléatoire et qui repose sur la clé du CPU. Cette clé d'enclave n'est jamais exposée en dehors de la portée de l'enclave, et donc personne d'autre n'est capable de l'inspecter pour déchiffrer les secrets pendant les communications.

2.3 SGX et dApps

La technologie SGX protège les applications off-chain de iExec tournant sur les nœuds distants (nommés *workers*) de façon que ces derniers ne pourront pas inspecter le code de l'application ou bien ses données secrètes. Le nœud *worker* peut seulement traiter les données chiffrées de cette application qui sont déchiffrées dans le CPU. Un attaquant qui a accès à cette machine même avec des privilèges ne peut pas accéder à ces données en clair. Également, le nœud *worker* ne peut pas modifier les données et le code chiffrés de l'application en question. Il peut seulement exécuter cette application et s'il essaye de l'altérer il ne peut plus l'exécuter puisque SGX détecte directement toute tentative de modification du code ou des données. Ces données sont chiffrées dans la mémoire par des clés dérivées de la clé du CPU qui ne peut pas être connue même par le possesseur de la machine. Par conséquent, c'est impossible de voler ces secrets en lisant ou inspectant la mémoire de l'application qui est en cours d'exécution. L'utilisation de la technologie SGX par iExec permet d'assurer que l'application en cours d'exécution sur le *worker* est correcte, que l'application s'exécute sans aucun changement (code et données) et que le résultat du calcul obtenu de son exécution est le vrai résultat attendu (pas inspecté par le nœud où elle est exécutée).

SGX protège de cette façon la confidentialité et l'intégrité des applications décentralisées de iExec. Cette technologie est emballée par la technologie SCONE qui est un conteneur sécurisé de Linux déployé dans le plate-forme de iExec pour protéger l'application distante. Je vais détailler plus SCONE dans la section 2.5.

2.4 Les conteneurs Docker et la sécurité

Je vais parler dans cette partie de la différence entre les machines virtuelles et les conteneurs Docker pour expliquer pourquoi iExec supporte Docker pour les dApps. Ensuite, je vais me concentrer sur le problème de sécurité dans Docker.

Les machines virtuelles sont une abstraction du matériel physique qui transforme un serveur en plusieurs serveurs. L'hyperviseur permet à plusieurs machines virtuelles de s'exécuter sur une même machine. Chaque machine comprend une copie complète d'un système d'exploitation, de l'application, des fichiers binaires et des bibliothèques nécessaires. Les machines virtuelles peuvent alors être lentes à démarrer.

Docker [16] est un outil conçu pour faciliter la création, le déploiement et l'exécution d'applications à travers l'utilisation des conteneurs. Ces derniers permettent à un développeur d'emballer une application avec toutes les bibliothèques et les dépendances qu'elle en a besoin, et de les regrouper en un seul *package*. De cette façon, l'application s'exécutera sur toute autre machine Linux, quels que

soient les paramètres personnalisés que la machine pourrait avoir et qui pourraient se distinguer de ceux de la machine utilisée afin de tester le code. Plusieurs conteneurs peuvent s'exécuter sur la même machine en partageant le même noyau du système d'exploitation, chacun exécutant des processus isolés dans l'espace utilisateur. C'est ce qui rend l'espace occupée par les conteneurs moins que celui occupé par les machines virtuelles.

Une similarité existe entre les conteneurs docker et les machines virtuelles quant à l'isolation et l'allocation des ressources. La différence principale entre les deux est que les conteneurs virtualisent le système d'exploitation au lieu du matériel ce qui rend docker plus rapide sachant que cette portabilité que fournisse docker est indispensable au monde du développement [17].

Concernant la sécurité, le noyau du système d'exploitation de la machine où tournent les conteneurs docker doit protéger une interface plus grande que celle à protéger dans une machine virtuelle. Les garanties d'isolation des conteneurs sont plus faibles que celles des machines virtuelles dans le noyau hôte du conteneur, car un hyperviseur fournit beaucoup moins de fonctionnalités qu'un noyau Linux (qui implémente généralement les systèmes de fichiers, les contrôles des processus applicatifs, etc.), sa surface d'attaque est beaucoup plus réduite.

La sécurité des conteneurs dépend des mécanismes purement logicielles utilisés. Ces mécanismes appliqués pour les conteneurs tournant sur une même machine protègent contre l'accès d'un conteneur non autorisé à un autre. Les données des applications tournant dans un conteneur ne peuvent pas être touchées par un autre conteneur même s'il partage avec lui le même système d'exploitation.

Cependant, les attaques qui peuvent avoir lieu sur les conteneurs peuvent être exercées par des tiers non autorisés non seulement par d'autres conteneurs. Le noyau du système d'exploitation ou bien l'hyperviseur qui sont des logiciels système privilégiés peuvent accéder à ces données ce qui permet par exemple à un compromis du OS de permettre d'attaquer l'application Docker et ses secrets [18].

2.5 SCONE : conteneurs sécurisés par Intel SGX

SCONE qui signifie en anglais : "Secure Container Environment" est une plateforme permettant de créer et d'exécuter des applications sécurisées par Intel SGX [19, 20]. L'objectif de SCONE est que toutes les données inactives, toutes les données sur le réseau ainsi que celles de la mémoire principale et même le code de l'application soient chiffrées. SCONE permet de protéger les données, les calculs et le code contre les attaquants même s'ils possèdent un accès privilégié (root) sur la machine où tourne cette application.

2.5.1 L'importance de SCONE dans le Cloud

Nombreux sont les avantages du Cloud pour les individus et les entreprises à la fois. L'utilisation des serveurs distants du Cloud pour traiter et stocker les don-

nées réduit les coûts d'investissement. Le fournisseur du Cloud utilise des machines virtuelles pour isoler les utilisateurs les uns des autres et de la machine puisqu'il ne ne peux pas faire confiance à ses clients. Ce qu'il faut prendre en compte aussi est que les informations stockées des utilisateurs dans les data centers du Cloud peuvent être confidentielles d'où vient la nécessité de protéger ces données des attaquants [21, 22]. En outre, l'application de l'utilisateur tourne sur une machine distante qui est administrée par le fournisseur du Cloud. Ce dernier peut être potentiellement malveillant donc il peut se logger sur les machines et inspecter les données des clients. Le risque de sécurité dans le Cloud est un sujet assez important et la nécessité de sécurisation est primordial dans cet environnement à cause du grand nombre des utilisateurs de cette technologie.

En outre, les conteneurs sont une solution au problème de la fiabilité du logiciel lorsqu'il est déplacé d'un environnement informatique à un autre. Comme le conteneur n'embarque pas d'OS, ce qui le différencie de la machine virtuelle, il est par conséquent beaucoup plus léger que cette dernière. Également, les conteneurs Docker, du fait de leur légèreté, sont portables de Cloud en Cloud. Les avantages des conteneurs Docker justifient l'importance de son utilisation dans le Cloud. Les principaux fournisseurs de Cloud (Microsoft Azure, AWS, Google, IBM, etc.) utilisent les conteneurs Docker dans leurs services offertes.

Personne ne peut nier le grand intérêt du Cloud et de l'utilisation des conteneurs Docker dans cet environnement. Le défi qui reste est comment exécuter les applications Docker dans un Cloud potentiellement non fiable de façon sécurisée. SCONE répond à ce problème en promettant d'assurer la sécurité des conteneurs Docker du Cloud par le *hardware* et de garder une performance acceptable pour l'exécution des applications. SCONE est un conteneur sécurisé pour Docker qui utilise la technologie SGX pour exécuter les processus des conteneurs dans des enclaves sécurisées et en les protégeant contre les attaquants même ceux ayant un accès root sur la machine où tourne l'application. SCONE peut chiffrer de manière transparente les fichiers et le trafic réseau et protège ainsi les données contre les accès non autorisés via le système d'exploitation et l'hyperviseur. Puisque SCONE utilise SGX, alors il permet à l'utilisateur de s'assurer que son programme tourne correctement sans aucune modification et avec toute sécurité dans le Cloud (attestation à distance).

SCONE fournit des images Docker organisées pour de nombreux services populaires tels que Memcached, MySQLApache, Nginx, etc. Le système d'exploitation doit inclure un pilote Linux pour SGX et un module Kernel pour SCONE.

2.5.2 Les défis

Les défis que les inventeurs du SCONE ont pris en considération dans leur implémentation sont :

1. L'interface externe : pour exécuter un programme dans un conteneur sécurisé, le conteneur doit avoir une librairie C standard (libc). L'implémentation de la libc utilise les appels systèmes qui ne sont pas permises dans l'enclave de SGX ce qui donne la nécessité d'exposer une interface externe au

système d'exploitation. Le défi pour SCONE est de choisir avec précaution quoi mettre dans cette interface et comment la protéger puisqu'avec SGX on ne fait pas confiance au OS et un attaquant qui accède au système d'exploitation peut utiliser cette interface exposée au OS pour compromettre les processus tournants dans le conteneur.

Haven [23], par exemple, est un projet de recherche qui vise à permettre l'exécution des applications Windows non modifiées dans le Cloud en toute sécurité sans besoin de faire confiance au fournisseur du Cloud. Haven, publié en 2015, utilise SGX pour assurer cette sécurité. Dans l'architecture de Haven, ils incluent toute la bibliothèque du OS Windows dans l'enclave. De cette façon, ils minimisent l'interface externe qui consiste en 22 appels systèmes uniquement car une grande partie des besoins des processus peuvent être fournis par la bibliothèque de l'OS qui est dans l'enclave. Ils introduisent la notion de *shielded execution* qui consiste à fournir des vérifications supplémentaires sur cette interface pour s'assurer si quelqu'un est entrain d'attaquer l'application à travers cette interface.

2. Un TCB minimal : le TCB (Trusted Computing Base) est l'ensemble des composants matériels et logiciels pouvant briser la politique de sécurité. C'est l'ensemble de composants à lesquels on va faire confiance en s'appuyant sur le principe qu'un petit TCB minimise la surface d'attaque. Par conséquent, il faut être sûr que ces composants ne sont pas malveillants. Dans le cas de SGX, on ne fait confiance qu'à l'enclave de l'application. Il faut alors décider quoi mettre dans cette enclave.

   Avec Haven, l'objectif n'étant pas de minimiser le TCB, mais plutôt que de donner à l'utilisateur la confiance de l'intégrité et de la confidentialité de ses données en les déplaçant d'un Cloud privé à un Cloud public. Cependant, ce TCB large dans Haven peut avoir potentiellement plusieurs vulnérabilités avec un impact sur la performance c'est pour cela que pour SCONE un des défis est de garder le TCB petit.

3. La performance des appels systèmes : avec SGX, les appels systèmes ne sont pas permises dans l'enclave ce qui affecte la performance de l'application (voir section 2.2.2). SCONE ont pris ce défi comme essentiel surtout pour le cas des applications qui utilisent beaucoup d'appels systèmes.

2.5.3 Les spécificités de l'architecture

Pour répondre aux défis déjà expliqués, SCONE a inclut dans son architecture ces caractéristiques :

1. Une petite TCB pour les conteneurs : pour gagner en performance et en sécurité, SCONE inclue une bibliothèque C qu'ils ont amélioré dans le TCB au lieu d'implémenter toute la bibliothèque OS qui est beaucoup plus grande (cas de Haven).
2. Des appels systèmes asynchrones : si une application construite avec SCONE veut effectuer des appels sytèmes, le *thread* de l'enclave n'a pas besoin de

sortir de cette dernière pour établir ces appels . D'abord, tous les informations reliées à ces appels systèmes (nombre des appels systèmes, les arguments, etc.) sont copiées dans une structure de donnée qui se trouve hors de la mémoire. Si on prend l'exemple d'une application qui veut lire des données, le programme a besoin d'un tampon pour lire ces données. Quand l'enclave effectue cet appel système, il ne faut pas que l'OS touche ce tampon (en retournant les données de l'appel système) d'où la nécessité d'avoir un autre tampon hors de l'enclave que SCONE a respecté. Pour établir la communication entre le Kernel et l'application afin d'effectuer les appels systèmes, une mémoire partagée est implémentée contenant deux files. Une des deux files est pour les requêtes et l'autre est pour les réponses de la part du Kernel. À chaque fois que l'application nécessite un appel système, l'indice du tampon est enfilé dans la file des requêtes et sera ensuite utilisée par le Kernel. Les *schedulers* de l'espace utilisateur vérifient ensuite si il y a une réponse dans la file remplie par le kernel (file de réponses aux demandes d'exécution des appels systèmes).

Également, SCONE est une bonne option pour les applications qui utilisent plusieurs *threads* puisqu'il permet à travers le *scheduler* de l'espace utilisateur de passer à exécuter d'autres appels systèmes avec d'autres *threads* en attendant la réponse de l'appel système d'un premier *thread* qui n'est pas encore réalisé.

2.5.4 L'intérêt de l'étude de la performance de SGX avec SCONE

Pour chaque nouvelle technologie, la performance représente un aspect essentiel à étudier. Même si une technologie assure des services étroitement alignés sur les besoins des utilisateurs, un élément important à considérer autre que son efficacité dans le domaine qu'elle serve est sa performance.

Les concepteurs de solutions de sécurité ont toujours débattu les compromis entre l'assurance de sécurité qu'ils fournissent et la performance qui peut être affectée à cause des méthodes de chiffrement utilisées.

SGX est une technologie de sécurité qui vise à protéger le code et les données secrètes des applications de toute modification ou divulgation. La clé de chiffrement manufacturée dans le *hardware* est le noyau de sécurisation de cette technologie. Cette clé et des clés dérivés de cette dernière protègent toute information confidentielle des clients de la blockchain, du Cloud et de l'IoT où on a toujours des problèmes de sécurité. D'ailleurs, il existe des facteurs dans SGX pouvant survenir des coûts et affecter la performance [24] tels que :

- La taille de l'enclave est limitée à 128MB puisque c'est une zone de confiance contenant le code sensible, elle devrait être aussi minimale que possible. Cette limite est imposée par le BIOS et c'est pour des raisons de sécurité.
- Les instructions privilégiées telles que les appels systèmes ne peuvent pas être exécutées dans l'enclave puisqu'on ne fait pas confiance au système d'exploitation. Les *threads* donc vont sortir de l'enclave et l'appel système

est exécuté par l'OS. Les entrées et les sorties de l'enclave et la gestion de l'enclave (pagination) ont un impact sur les performances (plus de délai pour l'exécution de l'application SGX que dans une exécution native, utilisation augmentée du CPU).

- Les bibliothèques de chiffrement de Intel n'optimisent pas la performance et la complexité de chiffrement et de déchiffrement de l'enclave.
- La nécessité de modifier l'application et de changer son code pour et de recompiler l'application de nouveau n'est pas pratique pour les applications qui ont beaucoup de code.

Avec la technologie SCONE, des changements au fonctionnement de SGX ont été appliqués tels que :

- L'utilisation des appels systèmes asynchrones (voir section 2.5.3) et de la mémoire partagée avec le kernel pour effectuer des appels systèmes hors de l'enclave en passant les paramètres nécessaires pour ces appels à travers cette mémoire. Cela élimine la nécessité d'effectuer des transitions par le *thread* de l'enclave (entrée et sortie de l'enclave) pour effectuer l'appel système en copiant à chaque fois l'état de l'enclave à chaque sortie et le reprenant à chaque entrée.

  SCONE permet de même au cas où une application a plusieurs *threads* qui vont utiliser des appels systèmes de permettre de servir les autres *threads* de l'application en attendant la réponse de l'appel système du premier sans nécessité de sortir de l'enclave.
- Les applications avec SCONE n'ont pas besoin d'être modifiées et elles peuvent être développées dans plusieurs langages tels que C++, JavaScript, Python, Java, Go, Rust, C, Fortan, etc.

Chapitre 3

Travail réalisé

Dans cette partie, je vais décrire la partie pratique de mon stage. Je commence par expliquer l'importance de l'étude de la performance de Intel SGX qui était le sujet de mon stage ensuite je passe à décrire les outils de mesure que j'ai utilisé. Plus tard, je me concentre sur les métriques que j'ai trouvé essentielles à étudier avec SGX. Je conclus par l'explication des applications utilisées pour mes tests avec une analyse des résultats.

3.1 Outils de mesure docker

J'ai fais d'abord une recherche sur les outils de mesure des applications Docker [25]. J'ai commencé par faire une comparaison entre quelques outils qui sont les plus fameux pour les tests des conteneurs Docker en réponse à la demande de mon tuteur. Ensuite, en effectuant les mesures et puisqu'il fallait que j'utilise des outils compatibles avec la technologie SCONE que j'ai adopté, j'ai cherché encore d'autres outils qui m'ont aidé à avoir plus de précision sur les résultats.

- Control groups (cgroups) sont des fonctionnalités du noyau Linux pour isoler l'utilisation des ressources (CPU, memory, disk I/O, et network) pour un ensemble de processus. Les conteneurs Docker et tous les conteneurs basés sur Linux reposent sur les *control groups*. Les mesures de CPU et des autres ressources pour les cgroups se trouvent dans la directoire `/sys/fs/cgroup`. Dans `/sys/fs/ cgroup /cu/docker/(id du conteneur)`, on peut voir, par exemple, la consommation de CPU par un conteneur qui est en cours d'exécution.

  Il existe deux moyens plus pratiques et plus faciles pour inspecter les statistiques d'utilisation des ressources par les conteneurs sans aucune installation :

  1. Docker stats : c'est un outil fourni par le client Docker [26]. Docker stats tourne dans Docker engine et interroge le système de fichiers virtuel "/cgroups" où les statistiques autour de la performance sont stockées. Pour consulter les statistiques du conteneur, il faut lancer la

commande docker stats avec le nom du ou des conteneurs en cours d'exécution pour lesquels on veut voir les statistiques. Cela présentera l'utilisation du processeur pour chaque conteneur, la mémoire utilisée et la mémoire totale disponible pour le conteneur. Et si pas de mémoire limitée pour les conteneurs, cette commande affichera la mémoire totale du hôte. Les stats sont mises à jour chaque seconde. L'option –no-stream peut être spécifiée quand on veut seulement que le premier *snapshot* des statistiques soit affiché et que les résultats ne sont pas en streaming. Également, on peut voir le total des données envoyées et reçues sur le réseau par le conteneur et le nombre de processus exécutés. Les statistiques Docker sont obtenus en temps réel et ne peuvent surveiller qu'un seul hôte.

2. Docker Remote API : pour avoir plus de métriques (i.e activité du disque) et plus de détails sur ces métriques, Docker Remote API peut être une bonne option. Docker daemon fournit une API à distance. Cette API est utilisée par le client pour communiquer avec Docker engine. Cette API peut être appelée par curl. L'API qui fournit des statistiques sur le conteneur est `/containers/id/stats` ou `/containers/name/stats`.

- cAdvisor : c'est une solution open source qui fournit des métriques d'hôte et de conteneur [27]. C'est un démon en cours d'exécution qui collecte, agrège, traite et exporte des informations sur l'exécution des conteneurs. Le *dashboard* de cAdvisor, qu'on peut y accéder via une page web, affiche les données pour les dernières 60 secondes seulement. Des outils tels que Prometheus et Grafana peuvent être utilisés avec cAdvisor. Prometheus est une base de données à indexation rapide récupérant les données des clients Cadvisor et Grafana crée des *dashboards* à lesquels on pourra avoir accès d'après une page web. Avec ces deux outils ont peut avoir un stockage, une récupération et une analyse à long terme des métriques de mesure. Cadvisor est simple à installer mais il consomme beaucoup de CPU.

- Librato : c'est un outil pour monitorer Docker [28]. Il inclue un ensemble complet de *dashboards* pour la visualisation des statistiques de mesure. Les données du conteneur qu'on veut monitorer en temps réel sont rassemblées et visualisées immédiatement. On peut afficher des statistiques pour tous les conteneurs Docker ou bien spécifier un seul conteneur et on peut filtrer en fonction du type de données généré pour un aperçu approfondi. Librato fonctionne en installant un petit agent (frais généraux négligeables) qui collecte des données et des métriques de système directement à partir du démon Docker exécuté sur le système. On peut spécifier quelles mesures spécifiques on veut collecter.

- Sysdig : c'est un service payant qui fournit le stockage, la visualisation et les alertes de métriques [29]. Il faut avoir un compte sur le site web de Sysdig

et remplir le formulaire d'inscription. Après cette étape, il faut indiquer ce qu'on veut monitorer et il faut exécuter une commande curl pour installer l'agent Sysdig qui a une option qui indique la clé secrète du client. C'est possible aussi d'installer un agent Docker Sysdig. On peut voir les mesures en temps réel ou voir une historique des données. La FIGURE 3.1 montre une comparaison générale entre les différents outils de mesure déjà mentionnés.

Outils Docker	Facilité d'utilisation	Graphe	Historique	Options supplémentaires	Overhead	Coût
Docker stats	OUI	NON	Temps réel seulement	Docker stat API	Non	gratuit
CADVISOR	OUI	Grafana	Avec les options	Prometheus Grafana	Utilise beaucoup de ressources	Open source et gratuit
Librato	Nécessité de configuration	OUI	OUI	----	négligeable	payant
Sysdig	OUI	Sysdig Monitor	OUI	SysdigCloud	négligeable	payant

FIGURE 3.1 – Une comparaison entre des outils de mesure Docker

Dans mes tests, j'ai choisi d'utiliser docker stats puisque j'ai voulu un outil qui est à la fois gratuit et ne consomme pas beaucoup de CPU. J'ai utilisé cet outil pour tester l'utilisation de CPU et de la mémoire par les applications Docker construites avec et sans SCONE .

Durant mes tests sur la performance, j'ai dû utiliser des outils autre que docker stats. J'ai essayé les mesures avec les outils suivants :

- strace : c'est un outil pour tracer les appels systèmes [30]. Il permet de visualiser les type des appels systèmes effectués par une application, leur nombre, le temps d'exécution de chacun de ces types d'appels en secondes, le temps total des appels systèmes et les erreurs s'ils existent. J'ai essayé cet outil sans SCONE après l'installer de la ligne de commande et j'ai pu visualiser les résultats. Avec SCONE, il y avait un problème de compatibilité avec strace et donc j'ai pas pu continuer à utiliser cet outil et j'ai cherché deux autres alternatives (syscount et time) qui peuvent le remplacer.

- syscount : c'est un outil qui donne des statistiques sur les appels systèmes sur tout le système ou bien pour un processus spécifique [31]. J'ai utilisé cet outil à la place de strace sachant qu'il ne donne pas le temps des appels systèmes. J'ai pu bénéficier de cet outil pour avoir le nombre d'appels systèmes pour chaque type de ces derniers.

- time : c'est une commande linux qui permet de déterminer le temps d'exécution d'une certaine commande [32]. Le résultat de cet outil est donné en trois paramètres :

(a) Real : c'est tout le temps écoulé, y compris les tranches de temps utilisées par le processus testé, le temps que le processus passe bloqué et le temps utilisé par les autres processus.

(b) User : c'est le temps d'exécution du processus testé uniquement. C'est plus précisément le temps qu'a pris le CPU en dehors du kernel et uniquement dans le code du mode utilisateur de ce processus.

(c) Sys : c'est le temps que le CPU a passé pour un processus spécifique et uniquement pour ce processus dans le mode kernel. Il représente le temps des appels systèmes effectués par ce processus.

Je me suis servie du paramètre Sys de la commande time pour avoir le temps total des appels systèmes effectués par les applications construites avec et sans SCONE.

- perf : c'est un outil d'analyse de la performance d'un processus [33]. Il est basé sur l'interface "perf_ events" qui est exportée par les dernières versions du kernel de Linux. perf_events est un outil qui collecte tous les événements qui ont eu lieu durant l'exécution du programme. Pour générer ces informations, j'ai utilisé la commande perf stat en indiquant le pid du processus à monitorer après l'option -p. La commande perf stat compte les événements concernant le programme et aussi ceux concernant le *hardware* telles que les instructions par cycle (IPC), le nombre de *cache misses*, ainsi que la fréquence du CPU.

  Quand l'application finit son exécution, l'affichage des événements est présentée dans la sortie standard. J'ai utilisé cet outil avec redirection de l'affichage dans des fichiers en indiquant le nom de fichier après l'option -o fournie par cette commande.

- top : j'ai utilisé top pour inspecter combien de mémoire physique et virtuelle est consommée par l'application Docker [34].

3.2 Métriques de mesure

Une métrique est un type de variable numérique utilisé pour évaluer une mesure quantifiable des performances d'un type d'objet tel que les applications dans mon cas. Les métriques peuvent être la charge du processeur, la consommation de la mémoire, le nombre de requêtes/seconde desservi par une application, la latence d'exécution, etc.

Pour les tests de performance que j'ai réalisé, j'ai étudié les métriques citées ci-dessous :

1. La consommation de CPU : toutes les données et le code stockées dans l'enclave de l'application sont chiffrées de façon transparente dans la mémoire physique par le MEE (Memory Encryption Engine). Cette unité matérielle

autonome utilise une combinaison compliquée des *Arbres de Merkle* (une clé de confidentialité de 128 bits), une version modifiée du AES (une compteur 56 bits), et une construction *Carter-Wegman MAC* (une clé d'intégrité de 128-bit qui produit des tags MAC de 56 bits et une clé de hash universelle de 512 bits pour la construction de MAC). Les clés utilisées par le MEE sont générées au démarrage et placées dans des registres du MEE et détruites quand le système devient en boot/sleep/hibernate c'est-à-dire à chaque fois qu'il est réinitialisé. Chaque chiffrement prend quatre opérations AES. La clé du MEE ne peut pas être accessible, elle est stockée dans le CPU.

À chaque fois que l'application a besoin de traiter des données, ses instructions sont chiffrées dans leur chemin vers le CPU et sont déchiffrées dans le cache du CPU. Les mécanismes de chiffrement et de déchiffrement utilisés engendrent une augmentation dans l'utilisation du CPU. C'est pourquoi, j'ai étudié cette métrique en premier lieu.

2. La latence d'exécution : c'est une métrique essentielle pour encourager ou non l'utilisation d'une application. Elle joue un rôle primordial dans la satisfaction de l'utilisateur. Une application mal optimisée avec des ralentissements visibles dans le temps de réponse peut entraîner un renoncement de l'utilisation de cette application par les clients.

   Avec la technique de chiffrement utilisée par SGX, c'est logique d'avoir un temps d'exécution de l'application qui est plus grand. Ce qu'il faut étudier c'est à quel point ce chiffrement peut affecter le temps de réponse.

3. La latence des appels systèmes : un processus peut avoir deux modes dans le système d'exploitation : mode utilisateur ou mode Kernel. Un programme ne peut pas accéder aux ressources de la machine (mémoire et matériel) s'il en a besoin en étant en mode utilisateur. Pour être capable d'utiliser ces ressources, le processus fait un appel au OS et donc il change de mode utilisateur en mode Kernel c'est ce qu'on appelle en anglais *context swicthing*. Ce type d'appel est connu sous le nom des appels systèmes qui sont des fonctions contenues dans le Kernel. Le processus accède au service demandé (manipuler des fichiers, créer des processus, etc.) à travers une API (Application Programming Interface).

   Avec SGX, l'exécution d'un appel système est un peu compliqué. On ne fait pas confiance au OS et donc les appels systèmes ne peuvent pas être émises depuis l'enclave. Si l'application a besoin des appels système, il faut les implémenter en appelant des fonctions en dehors de l'enclave. De cette façon, le *thread* de l'enclave va copier les arguments de la mémoire de l'enclave dans une mémoire à l'extérieur de l'enclave, sortir de l'enclave, exécuter une fonction pour lancer l'appel système et revenir à l'enclave après que l'appel système se réalise en copiant les résultats à cette mémoire chiffrée. En sortant et en revenant à l'enclave et pour ne pas avoir un risque

de sécurité, un ensemble de vérifications aura lieu avec un TLB (Translation Lookaside Buffer) flush. Le TLB est une cache du CPU qui stocke les dernières translations effectuées entre les adresses de la mémoire virtuelle et les adresses de la mémoire physique afin de réduire le temps pris pour accéder à la mémoire physique. La table des pages est une structure de donnée qui conserve les correspondances entre les adresses virtuelles et les adresses physiques pour un processus. Quand la table des pages change, le processeur vide systématiquement le TLB (TLB flush) ce qui peut être coûteux en terme de temps. Cela nous permet de dire que les appels systèmes avec SGX peuvent affecter la performance et ralentir l'application.

La technologie SCONE sert à la sécurisation des conteneurs Docker en utilisant SGX mais a trouvé une solution pour ce problème de délai d'appels systèmes rencontré avec SGX. SCONE fournit une interface pour les appels systèmes asynchrones [35]. Cette interface est l'interface externe exposée au OS mais protégée par des tests de vérification réalisés contre les attaques. Elle utilise une mémoire partagée pour permettre l'exécution des appels systèmes plus rapidement qu'avec SGX seul et d'une façon sécurisée. Deux files, une pour les demandes d'appels systèmes et une pour les réponses à ces demandes sont implémentées dans cette mémoire. Le *thread* de l'OS dans le module de kernel de SCONE traite ces appels et place les réponses sur ces appels dans la file de réponse.

Cette implémentation permet aussi de répondre aux demandes d'appels systèmes simultanées de plusieurs *threads* de la même application et non pas comme dans SGX sans SCONE où il fallait que tous les *threads* de l'application qui veulent utiliser des appels systèmes attendent la fin d'un appel système demandé en même temps par un autre *thread*.

4. La consommation de mémoire : la mémoire virtuelle consommée est toujours pour une application construite avec SGX autour de 64 GB, c'est la mémoire réservée pour l'enclave. Avec SGX v1, les enclaves doivent allouer toute la mémoire (64 ou 128 MB) au démarrage de l'application puisque les enclaves sont fixes. Le développeur de l'application peut changer la consommation de la mémoire en changeant les paramètres SCONE_HEAP et SCONE_STACK sachant que HEAP et STACK sont deux espaces mémoire, HEAP pour les variables dynamiques (dont on connaît le type et la taille à la compilation) et STACK pour les variables automatiques (dont on ne connaît le informations qu'à l'exécution). Dans la version 2 de SGX, ça serait possible d'allouer la mémoire à la demande et de changer d'une façon dynamique la gamme de l'enclave durant le temps d'exécution de l'application. Le temps de démarrage de l'application avec SGX peut être long si le développeur ne change pas manuellement les paramètres qui indiquent l'espace mémoire nécessaire pour une application.

5. Le coût de l'accès à la mémoire : comme la mémoire de l'enclave est toujours chiffrée, si on va copier ou lire des données dans l'enclave ça va être plus lent à cause du chiffrement et du déchiffrement qui aura lieu. Les deux cas suivants peuvent influencer sur le coût de l'accès à la mémoire :
   (a) Quand l'application a besoin d'écrire ou de lire de la mémoire, dès que des défauts de cache auront lieu il faut chercher les lignes de caches de la mémoire physique et donc chiffrer et déchiffrer les données déplacées entre la mémoire et le cache. Ce processus ajoute un coût (un délai) en comparant avec le fait de ne pas trouver les données dans le cache en exécutant une application sans SGX.
   (b) L'enclave a une taille limitée à 128MB. Lorsqu'une application a besoin de plus de mémoire physique, la pagination d'enclave peut être utilisée pour expulser certaines pages à échanger dans d'autres pages. Le CPU lit la page SGX à échanger, affiche le contenu de cette page et écrit la page chiffrée dans la mémoire physique non protégée. Plus le nombre de pages échangées est grand, plus les frais généraux augmentent et donc la performance sera affectée. De même, l'accès aux pages de l'enclave en dehors de la mémoire réservée cause des défauts de page.

3.3 Méthodologie des tests et résultats

Dans cette partie, je vais décrire la méthodologie des tests que j'ai effectué avec les outils utilisés. Je vais analyser chaque résultat obtenu de chaque test.

La machine que j'ai utilisé pour les tests est une machine du Cloud Intel Xeon E3-1230 v5 avec 4 cœurs et 8MB de cache. Le serveur a 16 GB de RAM et utilise Ubuntu 16.04.5 LTS avec la version 4.4 du Linux Kernel.

Pour construire une application avec SCONE, il faut indiquer dans le Dockerfile une image de SCONE à utiliser. Il faut également indiquer un paramètre en démarrant le conteneur qui lui indique d'utiliser le pilote de SGX, compiler l'application avec SCONE et choisir le mode *hardware* pour indiquer à l'application d'utiliser SGX. En définissant le paramètre SCONE_VERSION=1, on permet au développeur de l'application de s'assurer d'après l'affichage sur l'écran que son application s'exécute avec SGX en permettant d'afficher le *hash* de l'enclave (voir FIGURE 3.2).

(1) Test de la consommation de CPU

- L'application utilisée : j'ai utilisé une application de factoriel développée en python []. Cette application consiste à calculer le factoriel des nombres allant du plus simple au plus complexe. Le but d'effectuer ce test avec cette application est de détecter à quel point le calcul complexe peut augmenter l'utilisation du CPU avec SCONE comparant à l'exécution de cette même application dans un conteneur docker natif (sans SCONE).

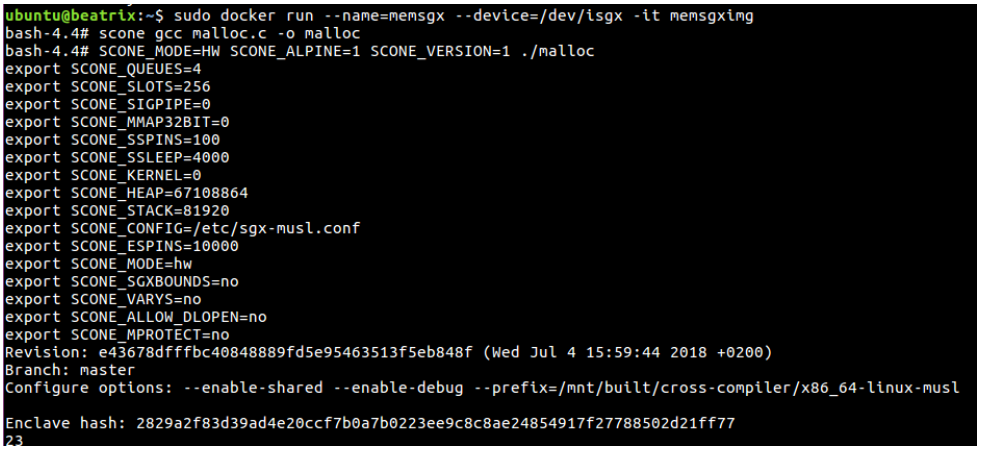

FIGURE 3.2 – L'exécution d'une application avec SCONE

- La méthodologie : après construire l'image Docker d'après une image SCONE et l'autre sans SCONE, j'ai automatisé les tâches à effectuer pour exécuter les conteneurs, réaliser les statistiques sur ces conteneurs en temps réel et ensuite afficher les résultats obtenus dans des graphes en conservant les statistiques obtenus dans des fichiers texte. Pour le réaliser, il fallait que j'exécute deux *threads* en même temps dont l'un a pour rôle de démarrer l'application dans un conteneur Docker et l'autre effectue les tests dans une boucle infinie sur cette application durant son temps d'exécution.

  Le deuxième *thread* est lancé par un script shell et dans ce dernier l'outil de test utilisé est l'outil natif de Docker qui est docker stats. J'ai bénéficié de cet outil afin d'avoir la consommation de CPU. J'ai programmé le second script de façon qu'à chaque seconde il affiche les statistiques et les ajoute à un fichier texte (voir FIGURE 3.3). Pour ce même script, j'affiche le pid à chaque démarrage. Ce pid me sert quand l'application s'arrête pour pouvoir tuer le shell qui effectue les tests. Et dès que l'application sera inactive, un programme en Python est appelée pour lire le fichier texte contenant toutes les mesures et pour dessiner les graphes représentant l'utilisation de CPU en fonction du temps en utilisant la bibliothèque Matplotlib [36].

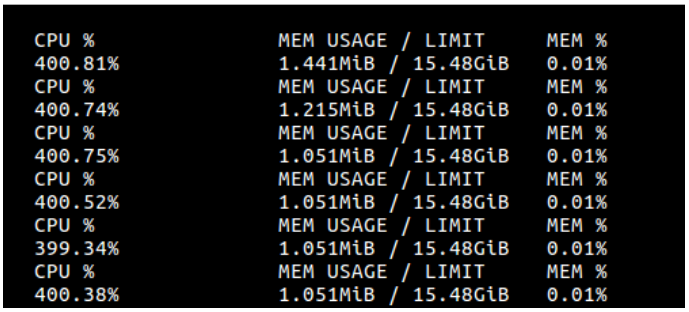

FIGURE 3.3 – Le résultat du test avec docker stats

- Les résultats obtenus : le fichier de configuration de SCONE contient des paramètres par défaut. Si ce fichier de configuration n'est pas modifié par l'utilisateur, c'est indiqué dans ce fichier que le nombre de threads utilisé dans l'enclave et pour servir les appels systèmes est plus grand que 1. J'ai effectué mes tests d'abord sans changer le fichier de configuration. Les résultats de la FIGURE 3.4 montrent que pour une application construite avec SCONE, la consommation de CPU reste égale à celle construite sans SCONE jusqu'au factoriel de 50 où cette consommation commence à augmenter jusqu'à atteindre 400% avec un factoriel de 100000 qui est 4 fois plus grand en comparant avec une application Docker native. Afin de comprendre ces résultats inattendus, j'ai contacté le responsable du site de SCONE qui m'a conseillé de changer le fichier de configuration de SCONE car le nombre de threads utilisés est assez grand par défaut avec cette technologie.

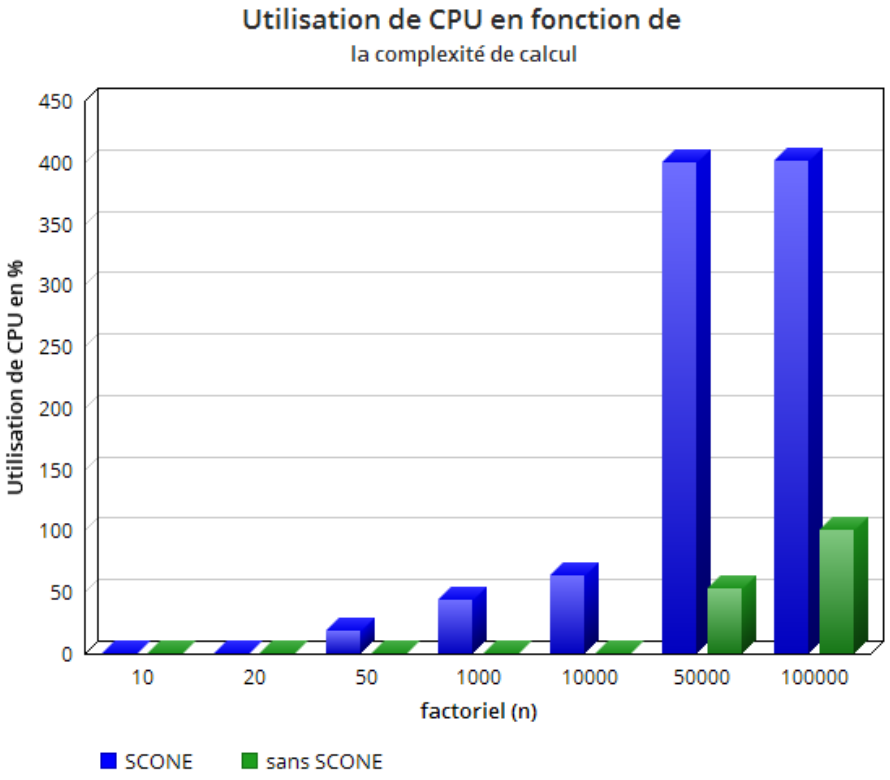

FIGURE 3.4 – L'utilisation de CPU avec/sans SCONE (plusieurs threads)

En réduisant le nombre de threads utilisé dans le fichier de configuration de SCONE, la consommation de CPU a changé. Dans mon cas, j'ai utilisé un seul thread dans l'enclave et un seul à l'extérieur de l'enclave.

Les résultats après ce changement étaient que l'utilisation de CPU est un peu plus élevée avec SCONE mais cette variation ne devient pas très remarquable qu'à partir d'un calcul de factoriel de 500000 où elle devient le double de celle sans

SCONE. Donc, on a atteint un résultat plus performant en diminuant le nombre de threads utilisé avec SCONE mais l'utilisation de CPU reste plus élevée qu'avec une execution native surtout en atteignant des calculs assez complexe (voir FIGURE 3.5).

Cette différence est expliquée par le chiffrement des données au cours de l'exécution de l'application avec SGX, et à cause de l'implémentation du module Kernel de SCONE qui utilise plus de threads pour exécuter les appels systèmes.

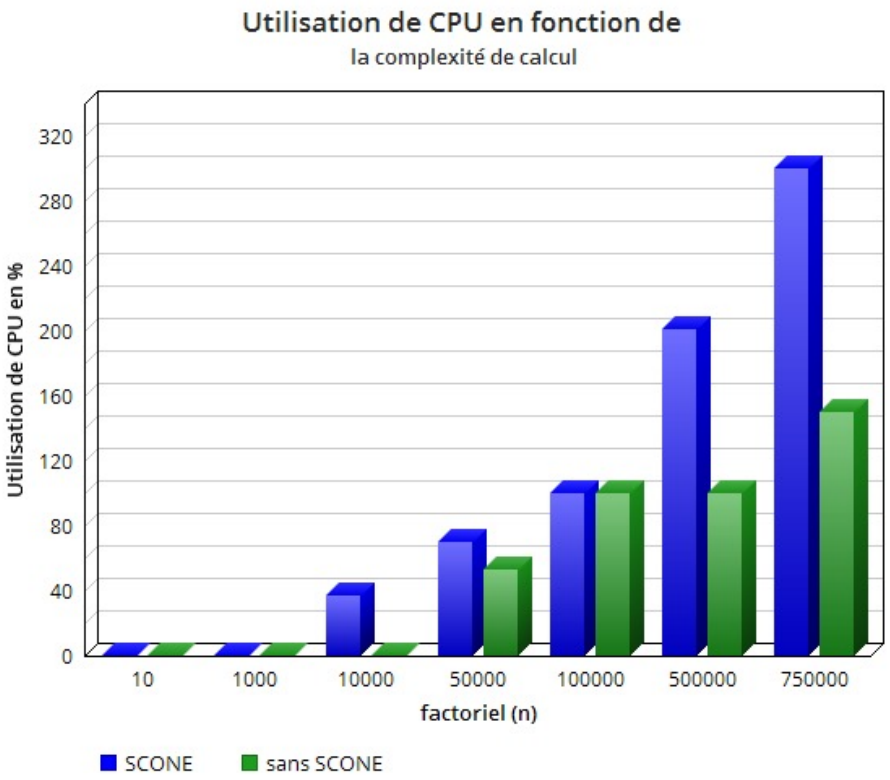

FIGURE 3.5 – L'utilisation de CPU avec/sans SCONE (un seul thread)

(2) Test du temps d'exécution

- L'application utilisée : avec l'application de factoriel aussi, j'ai étudié la variation de temps d'exécution. Le but d'effectuer ce test est de détecter à quel point le chiffrement utilisé dans SGX avec SCONE peut ralentir le fonctionnement d'une application.
- La méthodologie : l'outil docker stats affiche chaque seconde une mesure jusqu'à l'arrêt de l'application. Pour tracer directement la courbe, le programme python qui affiche les graphes de mesure de CPU compte aussi les lignes du fichier des résultats (chaque ligne paire représente une seconde) pour déduire le temps global de l'exécution de l'application.

- Les résultats obtenus : les résultats de la FIGURE 3.6 montrent que pour une application construite avec SCONE, la latence d'exécution commence à devenir remarquable à partir d'un calcul du factoriel de 100000. En calculant le factoriel des nombres supérieurs à 100000, la latence sera deux fois plus grande avec SCONE qu'avec une application Docker native. Par exemple, avec un factoriel de 750000, l'application prend 90 secondes à s'exécuter dans un conteneur Docker natif sachant qu'elle prend pour le même calcul 180 secondes avec le conteneur SCONE (voir FIGURE 3.6).

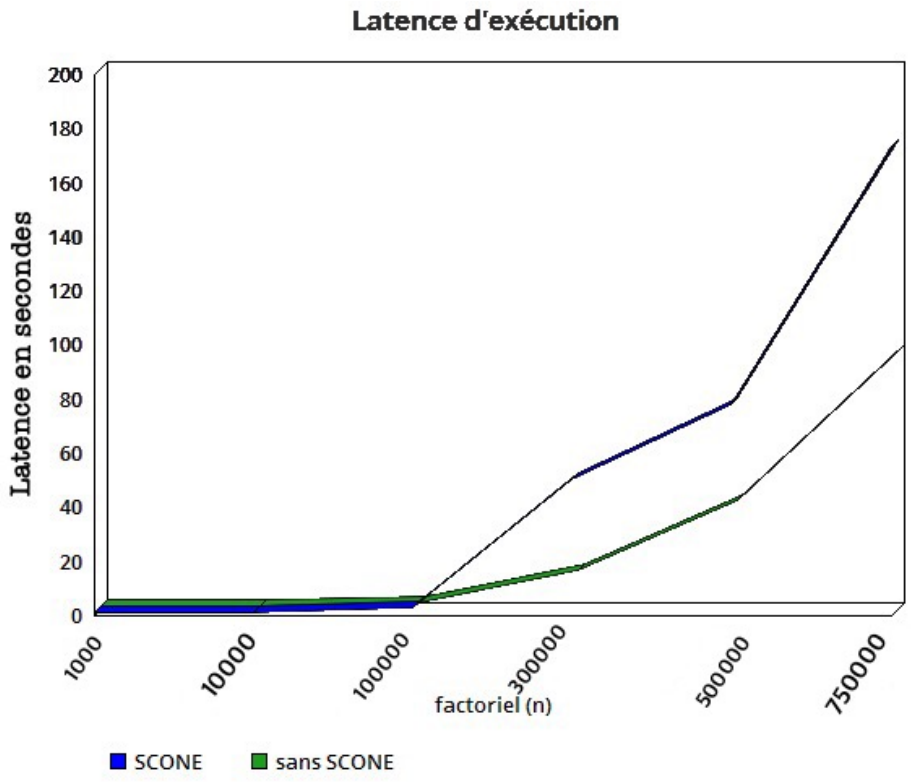

FIGURE 3.6 – Latence d'exécution d'une application avec/sans SCONE

(3) Test de la latence des appels systèmes

- L'application utilisée : j'ai créé une application développée aussi en python que j'ai nommée syscall.py.

  Cette application lit un fichier long de 1000 lignes dans une boucle d'un nombre n sachant que ce dernier est un entier qui est précisé par l'utilisateur. Dans la section 3.2, j'ai décris comment les appels systèmes sont exécutée avec SGX et la raison de la lenteur de leur execution. Les inventeurs de la technologie SCONE et même s'ils utilisent SGX par leur implémentation

de ce conteneur, mais ils ont amélioré le fonctionnement des appels systèmes. La lecture d'un fichier nécessite des appels systèmes. Le but de cette application est de tester combien de temps prennent les appels systèmes à s'exécuter en fonction de leur nombre.

- La méthodologie : j'ai utilisé la commande time dans ce test. Un des paramètres affichés dans le résultat de cette commande est la latence des appels systèmes effectués durant l'exécution de l'application.

  Je me suis servie aussi d'un autre outil qui est syscount avec les options -cp, -p pour indiquer le nom du processus qui effectue les appels systèmes et -c pour que je puisse avoir une vision claire du type d'appels systèmes effectués et de leur nombre.
- Les résultats obtenus : avec SCONE, le nombre d'appels systèmes est plus élevé qu'avec une application native. On peut remarquer que l'appel système utilisé par SCONE pour lire le fichier est *readv* à la place de *read* et le nombre de ce type d'appel système pour une boucle de 10 est 160 tandis que dans une application sans SCONE le nombre de *read* est 50.

  La FIGURE 3.3 montre les différents appels systèmes exécutés dans ce programme avec le conteneur sécurisé par SGX et avec un conteneur natif.

SCONE

exit_group	1
close	10
newfstat	10
readv	160
open	10

sans SCONE

exit_group	1
close	10
newfstat	20
read	50
open	10

FIGURE 3.7 - Les appels systèmes exécutés par le programme syscall .py

Les résultats présentés dans la FIGURE 3.8 montrent que pour une application construite avec SCONE, la latence des appels systèmes augmente avec le nombre d'appels systèmes réalisés d'une façon remarquable jusqu'à devenir 14 fois plus grand d'une exécution native avec une boucle de 200000 (6 200 000 appels systèmes).

Malgré les optimisations de SCONE pour réduire la latence des appels systèmes, le problème reste à cause de l'implémentation spécifique du module kernel de SCONE qui utilise des *threads* supplémentaires pour les appels systèmes ce qui ajoute un *overhead* sur la performance.

Des applications multi-thread peuvent bénéficier d'une exécution plus performante des appels systèmes puisque SCONE permet à plusieurs *threads* d'effectuer des appels systèmes simultanément sans besoin de faire attendre

tous les *threads* de l'application jusqu'à recevoir la réponse d'un appel système déjà demandée par un certain *thread*.

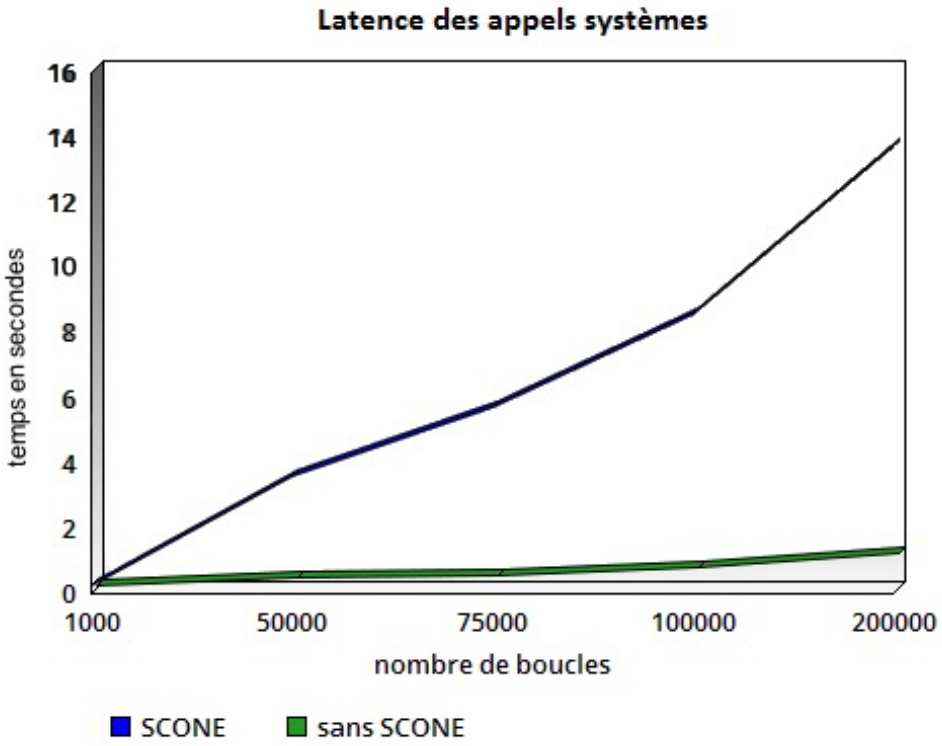

FIGURE 3.7 – Latence des appels systèmes d'une application avec/sans SCONE

(4) Test du coût de l'accès mémoire

- L'application utilisée : j'ai créé une application en C qui alloue dynamiquement de la mémoire avec la fonction malloc() (taille de bytes alloués indiquée par l'utilisateur), écris sur la mémoire allouée des entiers aléatoires et ensuite libère cette mémoire avec la fonction free(). J'ai voulu étudier le temps pris pour écrire sur la mémoire allouée dan l'enclave avec malloc() puisque la mémoire de l'enclave de SGX reste chiffrée ce qui affecte la performance d'une application qui a besoin d'accès mémoire excessif (écriture, lecture d'un grand nombre de bytes) et impacte la performance de l'allocation dynamique dans l'enclave.

  La mémoire virtuelle de SGX est de 64 GB et celle physique qui est l'enclave chiffrée est de 128 MB maximum. Ce qu'il faut savoir avant de commencer les tests est que la mémoire physique de l'enclave est déjà allouée durant le démarrage du programme, donc toute allocation dynamique dans l'enclave ne change rien puisque l'allocation est déjà faite. Si le programme a besoin

dans son allocation dynamique de plus d'espace allouée, il faut élargir la gamme des paramètres SCONE_HEAP et SCONE_STEAK.

- La méthodologie : j'ai utilisé l'outil perf stat qui m'a permis de savoir le nombre de défauts de cache, des détails sur les cycles de CPU, les instructions du programme exécutées et le IPC (Instructions Per Cycle). FIGURE 3.8 montre le résultat de la commande perf stat avec une application SCONE et une autre native en allouant une mémoire de 1024 bytes. Dans cette figure, on peut remarquer qu'avec SCONE et pour une allocation de 1024 bytes, le nombre d'instructions exécutées est énorme en comparant avec ceux exécutées sans SCONE et c'est à cause des instructions supplémentaires du CPU nécessaires pour contrôler l'enclave. Le temps d'exécution de tout le programme (task-clock) est de 9 secondes tandis que c'est de 0.2 msec sans SCONE.

  La fréquence d'horloge (clock cycles) est la vitesse à laquelle un microprocesseur exécute des instructions. Le CPU nécessite un nombre fixe de de cycles d'horloge pour exécuter chaque instruction. Plus l'horloge est rapide, plus le processeur peut exécuter d'instructions par seconde. La fréquence d'horloge est de 3.7 GHZ avec SCONE et de 1.55 GHZ sans SCONE donc le nombre d'instructions par cycles exécuté est plus élevée avec SCONE (IPC = 1.07 d'après la figure).

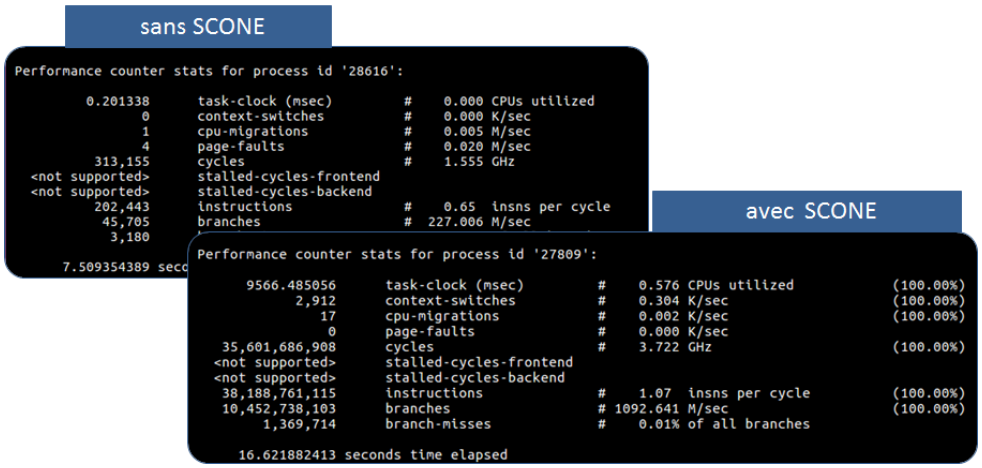

FIGURE 3.8 – Les résultats de perf pour une allocation avec/sans SCONE

Afin d'étudier le temps pris pour accéder à la mémoire et écrire les données, le temps pris par une application SGX dans son démarrage est plus grand en comparant avec une application native puisque dans ce temps de démarrage il faut allouer toute la mémoire de l'enclave pour l'application. Par conséquent, il ne faut pas considérer ce temps afin d'avoir un résultat précis pour cela j'ai utilisé clock(). Avec cette fonction, je sauve le temps avant l'écriture sur la mémoire puis je soustrais avec le temps après le traitement pour mesurer le temps d'exécution (voir FIGURE 3.9).

```
clock_t start = clock();
// Allocating memory with malloc  and wrting bytes on the size allocated
clock_t stop = clock();
double elapsed = (double)(stop - start) * 1000.0 / CLOCKS_PER_SEC;
```

FIGURE 3.9 – La méthode utilisée pour mesurer le temps d'accès mémoire

- Les résultats obtenus : la tâche d'allocation réalisée par la fonction malloc() et l'écriture dans cette mémoire allouée prend moins de temps avec SCONE quand le nombre de bytes écris est encore inférieur à 8MB qui est la taille de la cache L3 du CPU. La raison est que la mémoire est déjà allouée avec SCONE et donc il n'y a pas un besoin d'appels systèmes ou de demandes d'allocation ce qui rend ces opérations plus rapides qu'avec une application native qui a besoin d'allouer l'espace d'abord et ensuite l'utiliser.

  Tant que l'application demande une écriture sur une taille moins que 8MB, le temps d'exécution de l'application avec SCONE reste inférieur à celui sans SCONE. Dès que les défauts de cache auront lieu, le temps d'exécution commence à augmenter avec SCONE pour atteindre 110 millisecondes (le double du temps qu'a pris l'application native) avec une écriture de 80MB sur la mémoire. Le CPU cherche les lignes de caches de la mémoire physique c'est ce qui conduis à cette variation de temps d'exécution puisque le processus de chiffrement et déchiffrement des données déplacées entre le cache et la mémoire physique ajoute un coût. La FIGURE 3.10 montre le nombre de cache par défauts avec SCONE lors d'une écriture de 200MB sur la mémoire allouée (voir FIGURE 3.11 et 3.12).

```
ubuntu@beatrix:~$ sudo perf stat -p 5801 -e cache-misses
^C
 Performance counter stats for process id '5801':

         8,038,844      cache-misses

      12.795603970 seconds time elapsed
```

FIGURE 3.10 – Le nombre de défauts de cache lors d'une écriture mémoire

Ce qui est le plus remarquable est que lorsque la mémoire accédée sera supérieure à celle réservée pour l'enclave (64 MB est le HEAP_SIZE par défaut), le temps d'exécution devient 5 fois plus grand avec 400MB de données écrites par exemple où le délai est de 1723 ms avec SCONE et de 301 ms avec une application Docker native. Ce changement est dû au mécanisme de pagination qui aura lieu quand la mémoire de l'enclave n'est plus suffisante ce qui ajoute un coût au CPU pour lire les pages SGX qui doivent être expulsées, les chiffrer et les déplacer en mémoire non protégée. Il faut donc essayer de minimiser la taille de données dans l'enclave afin de garder une performance acceptable pour l'application (voir FIGURE 3.11 et 3.12).

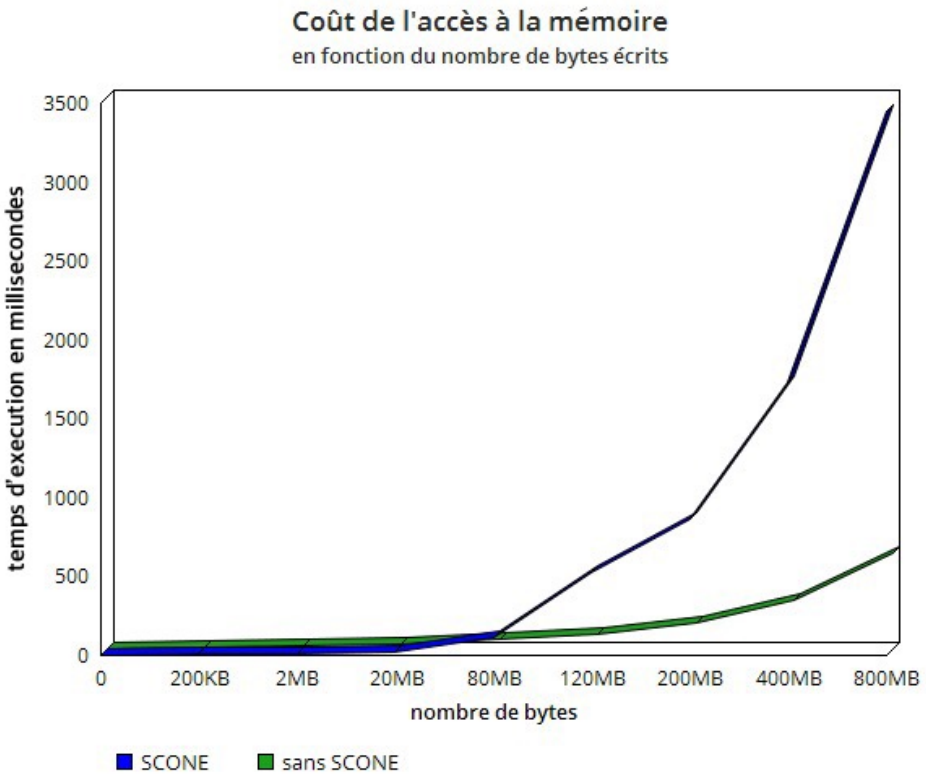

FIGURE 3.11 – La mesure du temps pris par l'écriture mémoire

taille de malloc	SCONE	sans SCONE
0	0	0
200KB	0.144	3.516
2MB	1.543	18.913
20MB	16	27.204
80MB	110.4	56
120MB	521.9	91.77
200MB	855	156.191
400MB	1723	301.399
800MB	3445	604.043

FIGURE 3.12 – La mesure du temps (en sec) pris par l'écriture mémoire

Chapitre 4

Apports du stage

Dans ce chapitre, je vais d'abord décrire les compétences techniques et relationnelles que j'ai acquises de ce stage. Ensuite, je soulignerai les difficultés rencontrées durant mon travail. Je finis par projecter l'expérience que j'ai reçue durant ce stage sur mon projet professionnel.

4.1 Compétences acquises

J'ai appris dans mon stage le langage de programmation Python qui est devenue très populaire à cause de sa simplicité. Puisque j'ai utilisé Docker dans tous mes tests, j'ai maintenant plus d'expérience sur cette technologie. J'ai acquis durant mon travail sur SCONE des connaissances sur la sécurité des conteneurs et j'ai appris qu'avec chaque technologie de sécurité il faut jamais oublier la performance. De plus, J'ai pris en main de Linux après être habituée à Windows puisque tout mon travail était sur une machine Ubuntu.

Au delà des compétences techniques acquises, ce stage m'a appris d'être autonome dans mon travail et de partager mes connaissances avec les autres dans l'entreprise. L'ambiance dans le bureau et dans toute l'entreprise en général était très conviviale, et j'ai pu développer des compétences relationnelles surtout qu'on était de différents nationalités (français, arabes, anglais, indiens, etc.) ce qui m'a permis d'échanger et de communiquer avec des collègues de différents cultures.

4.2 Problèmes rencontrées et solution apportées

Un des problèmes que j'ai rencontré dans mon stage était l'installation de SCONE sur ma machine. On a essayé moi et mon tuteur de l'installer mais il y avait un problème de ssh qui était nécessaire pour son fonctionnement.

Pour résoudre ce problème, j'ai utilisé une machine du Cloud où SGX était déjà installé. Cette machine a arrêté plusieurs fois de fonctionner et une fois le noyau était mis à jour sur cette machine et toute l'installation de SGX était supprimée. Quand j'avais pas accès à la machine du Cloud, j'ai profité de mon temps pour chercher sur mon sujet.

En outre, il y a quelques outils de mesure Docker qui ne sont pas compatibles avec SCONE et donc j'ai trouvé d'autres alternatives pour s'en sortir.

Le dernier problème était que j'ai commencé mon stage un peu tard et j'ai pris un mois de congé du travail pour les examens c'est ce qui m'a tardé de rendre le rapport de stage.

4.3 Projection de ce stage sur ma future carrière

Puisque je veux continuer en doctorat et dans un sujet concentré sur la sécurité par la cryptomonnaie, ce stage était pour mon bien. Mon expérience dans iExec Blockchain Tech m'a donné l'opportunité de candidater à des sujets de thèses sur la blockchain.

Également, SGX est une technologie impressionnante par son fonctionnement et le mécanisme de sécurité qu'elle utilise pour la protection des applications. Les informations que j'ai acquises sur la sécurité de cette technologie ont élargi mes connaissances et m'ont motivé de continuer dans le domaine de sécurité.

Chapitre 5

Conclusion

Pour conclure, j'ai effectué mon stage de fin d'études de mon Master en Informatique sur l'analyse de la performance d'une technologie de sécurité qui est Intel SGX au sein de l'entreprise iExec Blockchain Tech à Lyon. Lors de ce stage de 5 mois, j'ai utilisé mes connaissances théoriques et mes connaissances pratiques que j'ai appris durant ma formation à l'université sur Docker, Linux, la sécurité et le Cloud.

J'ai utilisé plusieurs nouvelles technologies qui sont assez importantes de nos jours tels que SGX, Docker et SCONE. Je me suis servie des informations que j'ai acquises durant ma recherche sur ces sujets pour tester et mesurer la performance de SCONE qui est un conteneur Docker sécurisé par SGX.

Les résultats que j'ai obtenu dans mes tests indiquent que l'utilisation de CPU, la consommation de la mémoire, le temps d'exécution et la latence des appels systèmes ainsi que le temps pris pour accéder à la mémoire sont plus élevés d'une façon remarquable en effectuant des calculs complexes avec SCONE qu'avec une application native. Pour améliorer cette performance, il faut essayer de minimiser tant que possible la taille de l'enclave afin d'éviter d'effectuer le chiffrement et le déchiffrement des données d'une façon excessive.

J'ai enrichi mon expérience durant ce stage sur Docker et ses tests de mesures, et j'ai pu étudier un nouveau langage de programmation qui est Python. J'ai pu avoir des relations d'amitié et de respect avec toute l'équipe de l'entreprise ce qui m'a permis d'améliorer mes capacités de communication et mon niveau de langue en ayant l'opportunité de discuter avec des collègues de plusieurs nationalités. La confrontation avec plusieurs problèmes durant mon alternance m'a entraîné à être responsable et organisée dans mon travail et de s'adapter à gérer les émotions dans le monde du travail. En plus, puisqu'en alternance le rythme est difficile à tenir, ce nouveau système de formation m'a appris de savoir gérer le stress et de se débrouiller dans les situations difficiles.

Cette expérience m'a motivé à choisir un sujet de thèse dans les domaines de la sécurité et de la cryptomonnaie. Ce stage m'a éclairé sur le choix des domaines où j'aime continuer à travailler et m'a beaucoup aidé pour mon projet professionnel.

Bibliographie

[1] iExec WhitePaper. iExec : A Blockchain-based Decentralized Cloud Computing . https://iex.ec/whitepaper/iExec-WPv3.0-English.pdf, , 2017-2018.

[2] Giang-Truong Nguyen and Kyungbaek Kim. A survey about consensus algorithms used in blockchain. *Journal of Information Processing Systems*, 14(1), 2018.

[3] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *Systems, Man, and Cybernetics (SMC), 2017 IEEE International Conference on*, pages 2567–2572. IEEE, 2017.

[4] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[5] Stimergy. https://stimergy.com/en/company/.

[6] Qarnot Computing. https://www.qarnot.com/fr/computing-ewerywhere-fr/.

[7] coinmarket. Coinmarketcap.com.

[8] dapp store. https://dapps.iex.ec/.

[9] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016 :86, 2016.

[10] Intel CORP. Intel Software Guard Extensions(Intel SGX,. https://software.intel.com/sites/default/files/332680-002.pdf, , June 2015.

[11] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost : Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News*, 42(1) :81–96, 2014.

[12] US DoD. Department of defense trusted computer system evaluation criteria (orange book). Technical report, Technical Report DoD 5200.28-STD, National Computer Security Center, 1985.

[13] Shiyong Lu Weidong Shi Saeid Mofrad, Fengwei Zhang. Memory Encryption Engine,). http://caslab.csl.yale.edu/workshops/hasp2018/HASP18_a9-mofrad_slides.pdf, , 2018.

[14] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017.

[15] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions : Epid provisioning and attestation services. *White Paper*, 1 :1–10, 2016.

[16] Dirk Merkel. Docker : lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239) :2, 2014.

[17] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[18] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor : retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

[19] F. Gregor T. Knauth A. Martin C. Priebe J. Lind D. Muthukumaran D. O'Keeffe M. L. Stillwell D. Goltzsche D. Eyers R. Kapitza P. Pietzuch S. Arnautov, B. Trach and C. Fetzer. Scone : Secure linux containers with intel sgx. In *OSDI*, volume 16, pages 689–703, 2016.

[20] scone docs. SCONE . `https://sconedocs.github.io/`.

[21] Ronald L Krutz and Russell Dean Vines. *Cloud security : A comprehensive guide to secure cloud computing*. Wiley Publishing, 2010.

[22] Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. On technical security issues in cloud computing. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 109–116. Ieee, 2009.

[23] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3) :8, 2015.

[24] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. Performance of trusted computing in cloud infrastructures with intel sgx. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science. Porto, Portugal : SCITEPRESS*, pages 696–703, 2017.

[25] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance : What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 11–16. ACM, 2017.

[26] docker docs. Docker stats. `https://docs.docker.com/engine/reference/commandline/stats/`, , June 2015.

[27] cadvisor . `https://github.com/google/cadvisor`.

[28] Librato . `https://www.librato.com/server-monitoring/docker`.

[29] Sydig . https://sysdig.com/opensource/.

[30] strace . https://linux.die.net/man/1/strace.

[31] syscount . https://github.com/brendangregg/perf-tools/blob/master/syscount.

[32] time . https://linux.die.net/man/1.

[33] perf . https://perf.wiki.kernel.org/index.php/Tutorial.

[34] top . https://linux.die.net/man/1/top.

[35] Livio Soares and Michael Stumm. Flexsc : Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 33–46. USENIX Association, 2010.

[36] matplotlib. https://matplotlib.org/.