ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE DES TECHNOLOGIES DE L'INFORMATION
M.Sc.A.

PAR
Manel ABDELLATIF

ACCÉLERATION DES TRAITEMENTS DE LA SÉCURITÉ MOBILE AVEC LE CALCUL PARALLÈLE

MONTRÉAL, LE 31 MARS 2016

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Chamseddine Talhi, directeur de mémoire
Département de génie logiciel et TI à l'École de technologie supérieure

M. Abdelwahab Hamou-Lhadj, codirecteur
Département de génie électrique et génie informatique à l'Université Concordia

Mme. Nadjia Kara, présidente du jury
Département de génie logiciel et TI à l'École de technologie supérieure

M. Sègla Kpodjedo, membre du jury
Département de génie logiciel et TI à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE "15 MARS 2016"

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je profite de cette occasion pour exprimer ma reconnaissance et adresser mes remerciements les plus sincères à toute personne qui m'a aidée et qui a contribué à ce projet de recherche.

Mes remerciements s'adressent également à mon directeur de mémoire, le professeur Chamseddine Talhi qui a bien voulu diriger ce mémoire tout en m'accordant sa confiance. Ses remarques pertinentes, son soutien appréciable et ses conseils judicieux ont beaucoup contribué à améliorer la qualité de ce mémoire.

Je tiens à remercier mon codirecteur de mémoire le professeur Abdelwahab Hamou-Lhadj, pour sa disponibilité et son support.

Je tiens aussi à remercier M. Michel Dagenais, pour sa collaboration, le temps qu'il m'a accordé et les efforts qu'il a fournis dans les travaux de recherches de ce mémoire.

J'exprime mes vifs remerciements et mon respect aux membres de jury, Mme Nadjia Kara, et M. Sègla Kpodjedo d'avoir accepté d'évaluer mon projet de recherche.

Je remercie aussi mes collègues du Laboratoire de Génie Logiciel GÉLOG et LASI. Je pense notamment à Abdelfattah Amamra, Tarek Slaymia, Ghassen Zagden, Oussama Boudar ainsi que mes chères Amel Ferjani et Ibtihel Amara.

Je dédie ce mémoire à ma famille, principalement à mes parents Habib et Samira, mon frère Iheb, ma sœur Siwar, ma belle-soeur Manel et particulièrement ma grand-mère Rachida. Je les remercie tous pour leur apport moral, leur amour, leur soutien et surtout leurs sacrifices.

ACCÉLERATION DES TRAITEMENTS DE LA SÉCURITÉ MOBILE AVEC LE CALCUL PARALLÈLE

Manel ABDELLATIF

RÉSUMÉ

L'accélération des traitements relatifs à la sécurité mobile est devenue l'un des problèmes les plus importants vu la croissance exponentielle et l'impact important des attaques ciblant ces plateformes. Il est important de protéger les informations sensibles au sein des téléphones mobiles à travers l'implantation de systèmes de détection de malwares ainsi que le chiffrement des données dans le but de maintenir un plus haut niveau de sécurité. En effet, pour détecter les applications malveillantes, un antivirus analyse un flux de données important et le compare avec une base de données de signatures de malwares. Malheureusement, comme le nombre de menaces augmente continuellement, le nombre de signatures de codes malveillants augmente proportionnellement. Ceci rend le processus de détection plus complexe pour les téléphones mobiles, surtout qu'ils sont limités en termes de mémoire, de batterie et de capacité de traitement. Comme le niveau de sécurité de ces systèmes s'aggrave, la capacité de calcul parallèle pour les téléphones mobiles est de mieux en mieux améliorée avec l'évolution des unités de traitement graphiques mobiles (GPU).

Dans ce mémoire, nous allons porter l'accent sur comment nous pouvons tirer profit de l'évolution des capacités de traitement parallèle des appareils mobiles afin d'accélérer la détection des logiciels malveillants ainsi que les traitements de cryptographie sur les téléphones Android. Dans ce but, nous avons conçu et mis en œuvre une architecture parallèle pour les appareils mobiles qui exploite les capacités de calcul des GPUs mobiles et le traitement distribué sur les clusters. Une série de techniques de calcul et d'optimisation de la mémoire est proposée pour augmenter l'efficacité de la détection et le débit d'exécution.

Les résultats de ce travail de recherche nous mènent à conclure que les GPUs mobiles peuvent être utilisées efficacement pour accélérer la détection des malwares pour les téléphones mobiles ainsi que les traitements cryptographiques. Les résultats montrent également que l'architecture locale proposée sur les téléphones mobiles peut être étendue à une architecture de cluster afin d'avoir un taux d'accélération de traitement plus important lorsque les ressources du téléphone mobile sont occupées.

Mots clés: Détection de malwares, Cryptographie, Calcul parallèle, Correspondance de patrons, Analyse des traces des téléphones mobiles, Cluster

ACCELERATION OF MOBILE SECURITY PROCESSING WITH PARALLEL COMPUTING

Manel ABDELLATIF

ABSTRACT

Accelerating mobile security processing is becoming one of the most challenging problems. It is important to protect sensitive information in mobile phones through malware detection systems and data encryption to maintain a high security level. In fact, malware detection on mobile phones involves analyzing and matching large amount of data streams against a set of known malware signatures. Unfortunately, as the number of threats grows continuously, the number of malware signatures grows proportionally. This is time-consuming and leads to expensive computation costs, especially for mobile devices where memory, power and computation capabilities are limited. As the security threat level is getting worse, parallel computation capabilities for mobile phones is getting better with the evolution of mobile graphical processing units (GPUs).

This thesis focuses on how we can get benefit from the evolving parallel processing capabilities of mobile devices in order to accelerate malware detection as well as cryptographic processing on Android mobile phones. For this purpose, we have designed and implemented a parallel architecture for mobile devices that exploits the computation capabilities of mobile GPUs and distributed processing on clusters. A series of computation and memory optimization techniques are proposed to increase the detection and processing throughput.

The results suggest that mobile graphic cards can be used effectively to accelerate malware detection for mobile phones as well as cryptographic processing. The results show also that the local processing on mobile phones can be extended to cluster architecture in order to have more interesting processing acceleration rates when the mobile phone is busy.

Keywords: Malware detection, Cryptography, Parallel processing, Multi-pattern matching, Trace analysis of mobile phones, Cluster

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

AES	Advanced Encryption Standard
AC	Aho-Corasick
COPRTHR	CO-PRocessing THReads
CPU	Central Processing Unit
CW	Comment-Walter
DA	Detection Accuracy
DES	Data Encryption Standard
ECC	Eliptic Curve Cryptography
ELF	Executable and Linking Format
ESDK	Epiphany Software Development Kit
ETS	École de Technologie Supérieure
FN	False Negaitive
FP	False Positive
GPU	Graphics Processing Unit
MPI	Message Passing Interface
PVM	Parallel Virtual Machine
STM	State Transition Matrix
TN	True Negaitive
TP	True Positive
WM	Wu-Manber

INTRODUCTION

Depuis plus de deux décennies, la performance des microprocesseurs basés sur une seule unité de traitement a augmenté rapidement ce qui a engendré des réductions de coûts considérables dans les systèmes informatiques. Par conséquent, plus de fonctionnalités et de meilleures interfaces utilisateurs ont été fournies. Cependant, l'accélération des traitements informatiques qui est liée à l'augmentation de la fréquence d'horloge des processeurs a connu rapidement des limites, vu la dissipation d'énergie excessive et le problème de réchauffement de ces unités de traitement. Les utilisateurs, à leur tour, demandent un niveau supérieur de performance de calcul. Sans amélioration de la performance, les développeurs d'applications ne seront plus en mesure d'introduire de nouvelles fonctionnalités et des capacités de calcul dans leurs logiciels.

Pour cette raison, plutôt que de continuer à accroître la taille et la vitesse des processeurs, les concepteurs ont cherché progressivement à paralléliser les tâches à exécuter et à les confier à des architectures multicœurs. Ainsi, le calcul parallèle est devenu de plus en plus répandu et demandé surtout avec le développement de plusieurs types de composants électroniques offrant ce type de calcul comme les cartes graphiques (GPU), les DSP, FPGA, etc. Le développement de la programmation parallèle a rendu possibles plusieurs types de traitements dans des domaines variés comme le traitement d'image, la cryptographie, l'intelligence artificielle, la sécurité, etc.

La montée en puissance de la programmation parallèle a touché aussi les systèmes embarqués. En effet, la complexité logicielle et matérielle des systèmes embarqués, notamment les appareils mobiles, ne cesse d'augmenter. Par ailleurs, le calcul parallèle est devenu populaire dans ces plateformes grâce à l'intégration des GPUs mobiles de plus en plus sophistiquées. Depuis quelques années, des sociétés comme ARM, NVIDIA et Qualcomm se sont intéressées par ce marché et ont investi dans la conception des GPUs mobiles dont la performance ne cesse d'augmenter. Cette tendance est apparue vu les enjeux et l'envergure de l'utilisation des téléphones intelligents. Les capacités révolutionnaires du calcul graphique en 2D et 3D au niveau des applications mobiles sont rendues possibles grâce aux performances de plus en plus améliorées des GPUs mobiles.

Cependant, la sécurité logicielle des systèmes embarqués, plus particulièrement les téléphones intelligents, est devenue une préoccupation de plus en plus importante vu la croissance exponentielle et l'impact des attaques ciblant ces plateformes. En effet, un grand nombre d'utilisateurs de toutes sortes utilisent au jour le jour les téléphones dits "intelligents" comme moyen de communication, d'organisation, et de gestion de leur vie professionnelle et privée. Par ailleurs, les téléphones mobiles sont devenus une source de risques potentiels puisqu'ils collectent une quantité importante d'informations considérées sensibles et dont l'accès doit être contrôlé afin de sécuriser la vie privée de l'usager. Tout comme les ordinateurs, les smartphones sont des cibles privilégiées d'attaques. Ces attaques exploitent plusieurs faiblesses liées au smartphone : cela peut provenir des moyens de communication comme les SMS/MMS et les réseaux Wi-Fi et GSM. Ensuite des vulnérabilités aux attaques exploitant les failles logicielles qui peuvent provenir aussi bien du navigateur web que du système. Kaspersky, l'une des plus grandes firmes de service de sécurité estime que 99% des attaques détectées au niveau des téléphones mobiles en 2013 ciblaient la plateforme Android et qu'en 2012 le nombre d'attaques survenues sur le marché du mobile a augmenté de 163%.

C'est ainsi que la sécurité logicielle des smartphones est devenue une préoccupation de plus en plus importante de l'informatique liée à la téléphonie mobile. Elle est particulièrement préoccupante, car elle concerne la sécurité des informations personnelles disponibles au sein des Smartphones. Il existe une panoplie de contre-mesures ainsi que d'approches adoptées pour assurer la sécurité des téléphones intelligents. Cependant, on est toujours contraint aux performances limitées de ces téléphones à savoir la mémoire, le processeur, la batterie, etc.

Par ailleurs, pour détecter les logiciels malveillants, un antivirus analyse un flux de données important et les compare avec une base de données de signatures de malwares. Malheureusement, le nombre des signatures des logiciels malveillants augmente proportionnellement avec le nombre important d'attaques. Ceci rend le processus de détection plus complexe pour les téléphones mobiles, surtout qu'ils sont limités en termes de mémoire, de batterie et de capacité de traitement. Parmi les solutions communes pour remédier à ce problème est l'envoi des données à analyser à un serveur externe. Cependant, cette solution rend la sécurité du téléphone

mobile dépendante du serveur externe et l'expose aux problèmes de connectivité. L'intégration d'un outil de détection de malware au sein du téléphone mobile est donc indispensable pour assurer plus de sécurité. Les systèmes cryptographiques offrent également une autre couche de sécurité à travers le chiffrement des données échangées ou stockées au niveau des appareils mobiles. Cependant, les protocoles cryptographiques sont également complexes et gourmands en temps de calcul ce qui augmente la nécessité de les optimiser pour ce type de systèmes.

Ainsi, au niveau de l'environnement des téléphones mobiles nous avons deux principaux éléments : le niveau de la sécurité des appareils mobiles qui s'aggrave d'une part, et les capacités de traitement parallèle améliorées avec l'évolution des cartes graphiques mobiles d'autre part. Les questions de recherche qui se posent à ce niveau est : comment peut-on donc accélérer les traitements de la sécurité mobile avec le calcul parallèle ? Quelle technique de détection peut-on utiliser pour assurer une bonne précision de détection ? L'objectif principal de cette recherche est la définition d'une plateforme d'accélération des traitements de la sécurité mobile à savoir la détection de malwares et la cryptographie, à travers le calcul parallèle tout en utilisant des systèmes à ressources limitées.

Cet objectif peut être résumé en cinq sous-objectifs :

- Exploiter les techniques d'accélération du traitement parallèle sur les cartes graphiques mobiles ;
- Cibler une technique de détection de malwares sur les téléphones mobiles offrant une bonne précision de détection ;
- Exploiter les techniques de compactage de données pour remédier à la contrainte de la mémoire réduite offerte par les systèmes embarqués ;
- Cibler un algorithme de cryptographie qui peut être supporté et accéléré par les GPUs mobiles ;
- Étendre l'architecture proposée sur un cluster pour assurer une plus grande performance de calcul et le recours aux plateformes hautement parallèles dans le cas de la non-disponibilité des ressources sur le téléphone.

La méthodologie que nous proposons pour atteindre ces objectifs est la suivante. D'abord, l'exploration du domaine de recherche et ce, en se basant sur la revue de littérature. Ensuite la conception d'une plateforme pour l'accélération des traitements de la sécurité mobile exploitant une bonne technique de détection et intégrant également un module de cryptographie parallèle. Puis la définition d'une approche de validation à travers les expérimentations.

Ce mémoire se compose de quatre chapitres. Dans le premier chapitre, nous présentons les notions de base relatives aux calculs parallèle et distribué ainsi que les traitements de cryptographie et les méthodes de détection de malwares mobiles. Dans le deuxième chapitre nous présentons la revue de littérature sur l'accélération du pattern matching avec le calcul parallèle ainsi que les travaux sur la cryptographie parallèle. L'utilisation de l'architecture de cluster pour la sécurité est également abordée dans ce chapitre. Le troisième chapitre décrit l'architecture générale de l'approche proposée ainsi que les différentes techniques d'optimisation que nous avons utilisées pour accélérer l'exécution des traitements de notre architecture. Enfin, dans le quatrième chapitre une étude expérimentale et une validation de l'approche proposée sont abordées pour justifier les choix que nous avons pris. Finalement, une conclusion générale de ce mémoire est présentée contenant une synthèse du travail présenté ainsi que ses différentes limites et perspectives.

CHAPITRE 1

NOTIONS DE BASE

1.1 Introduction

L'accélération des traitements de la sécurité mobile est devenue une préoccupation de plus en plus importante vu le développement de la capacité de calcul parallèle dans ce type de systèmes. Pour aborder ce problème, il est important d'introduire quelques notions de base nécessaires à la compréhension et à la résolution de notre problématique de recherche. Dans ce but, nous allons étudier l'évolution des systèmes embarqués notamment les téléphones mobiles tout en mettant l'accent sur l'évolution des GPUs mobiles qui représentent la principale cible de notre travail de recherche. Ensuite, une introduction sur les paradigmes de calcul parallèle sera abordée ainsi que les techniques de détection de malwares mobiles et le besoin d'optimisation par le parallélisme. Enfin, une description de quelques algorithmes de correspondances de patrons sera abordée ainsi que quelques algorithmes de cryptographie.

1.2 Les cartes graphiques

Durant ces dernières années, la conception du matériel électronique pour les systèmes embarqués notamment les téléphones mobiles a été l'un des axes les plus dynamiques du marché de la technologie moderne. L'utilisation importante des téléphones intelligents et la tendance à la hausse des ventes de ces appareils à l'échelle mondiale ont attiré des entreprises de partout dans l'industrie pour essayer de produire du nouveau matériel et des logiciels pour ces appareils. Bien que les processeurs RISC ARM aient été les processeurs principaux utilisés à l'intérieur de ces dispositifs, avec le système Android de Google et le système d'exploitation iOS d'Apple, il y a encore plusieurs entreprises concurrentes du marché du GPU mobile. Les sociétés AMD, ARM, Nvidia, PowerVR et Qualcomm sont considérées comme leaders de l'industrie des GPUs mobiles. Nous nous pencherons alors sur quelques-unes d'entre elles pour avoir un aperçu de ce domaine très concurrentiel de l'industrie du smartphone.

D'une façon générale, les premières cartes graphiques ont été développées dans les années 1980 pour diminuer la charge allouée aux processeurs pour l'affichage des textes et le graphisme en général. Elles représentent des unités de traitements secondaires connectées aux CPUs et dotées de leur propre espace mémoire. Avec la demande accrue en matière de graphisme, d'affichage d'un nombre de plus en plus important de pixels ainsi que l'utilisation de jeux vidéo de plus en plus développés, les GPUs ont commencé à se perfectionner. Ce sont les jeux vidéo qui ont permis la démocratisation des GPUs en apportant la diffusion en masse et donc la baisse des coûts que ce soit pour les ordinateurs ou les appareils mobiles. L'importante demande en matière de graphisme a donc permis l'évolution des cartes graphiques mobiles. Comme nous le pouvons remarquer dans la figure 1.1, le marché des GPUs mobiles est en perpétuelle évolution. Cette évolution de performances est estimée de 40% chaque 4 ans.

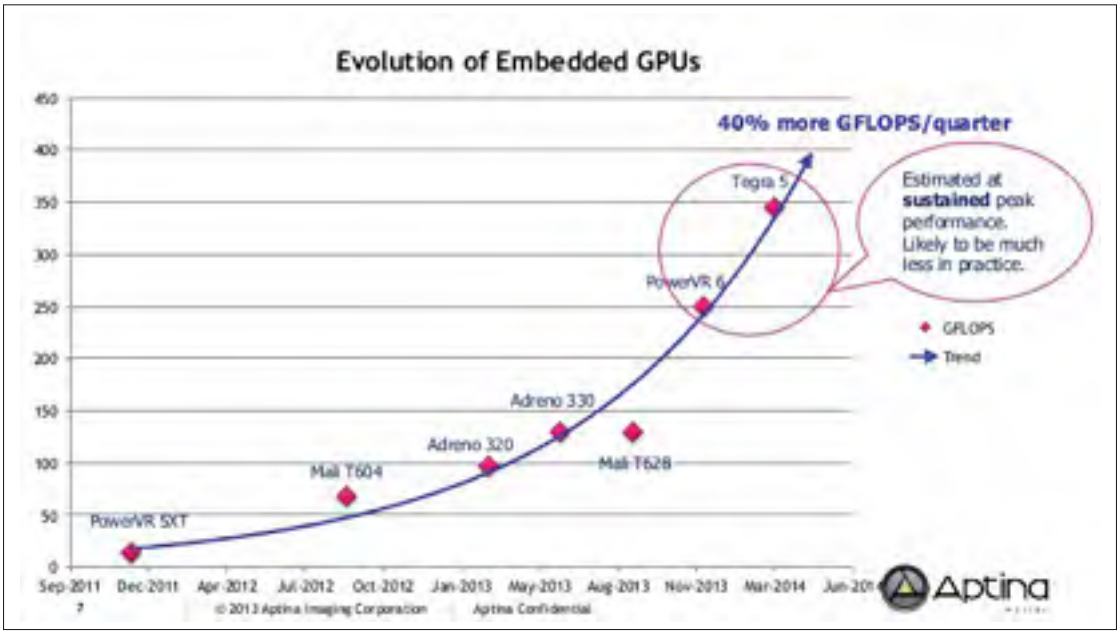


Figure 1.1 Evolution des GPUs dans les systèmes embarqués
Tirée de Bourges-Sevenier (2013)

Mieux encore, la performance de la série des GPUs Mali d'ARM par exemple a doublé cinq fois durant ces deux dernières années. Ces GPUs sont conçues pour produire du calcul à haute performance tout en maintenant une efficience énergétique pour répondre aux contraintes limi-

tées des téléphones mobiles, des tablettes ou encore des télévisions intelligentes dans lesquels elles sont implantées. ARM introduit de plus en plus de modèles de GPUs mobiles comme par exemple Mali T-830 à quatre cœurs qui offre une performance meilleure de 55% par rapport à son prédécesseur Mali-T622 avec une surface en silicone plus réduite (ARM, 2015). Les GPUs de Qualcomm sont considérés comme leaders dans le graphisme des tablettes et des téléphones mobiles. En effet, en 2015 la GPU Adreno 430 présente principalement dans les téléphones mobiles Xperia Z4 a été classée comme meilleure GPU mobile pour les téléphones Android vu les performances qu'elle offre (Qualcomm, 2015).

Nvidia possède également sa part de marché des GPUs mobiles. Les GPUs Nvidia sont plutôt répandues sur les ordinateurs de bureau. Parmi les modèles les plus récents de Nvidia dans le domaine de l'embarqué on peut citer la GPU de Tegra 4 dotée de 72 cœurs et qui fait l'alliance entre puissance et rendement énergétique. Un autre modèle intéressant de Nvidia pour la plateforme mobile est la GPU Nvidia Maxwell à 256 cœurs qu'elle vient de lancer avec les processeurs mobiles Tegra X1. Cette GPU est 50% plus rapide que son prédécesseur Tegra K1 offrant une puissance maximale allant jusqu'à 1 téraflops (Nvidia, 2015).

Ainsi, d'après cette étude non exhaustive de la performance des GPUs mobiles, nous pouvons constater que ce marché est devenu assez mûr pour offrir des outils de calcul parallèle permettant d'accélérer le traitement des données vu les performances intéressantes que les GPUs offrent tout en prenant en considération les différentes contraintes de l'environnement mobile (mémoire réduite, consommation d'énergie, etc.).

1.3 OpenCL

Plusieurs sont les plateformes de programmation parallèle sur les GPUs. Parmi ces plateformes on peut citer Cuda (supporté uniquement pour les GPUs Nvidia), OpenGL (pour les traitements graphiques), Renderscript (pour les systèmes Android) et enfin OpenCL. Ce dernier a été conçu par le groupe Khronos en 2009. La portabilité de OpenCL sur différents types d'unités de

traitement et sa rapidité d'exécution constituent les deux majeurs avantages de cette plateforme de calcul parallèle.

Comme c'est illustré dans la figure 1.2, OpenCL distingue deux types d'éléments : le processeur hôte (généralement la CPU) et ses périphériques (GPU, FPGA, Co-processeur Epiphany, etc.). Le processeur hôte joue le rôle d'orchestration des tâches à exécuter en parallèle. Les périphériques quant à eux exécutent les modules de calcul parallèles appelés *Kernels*.

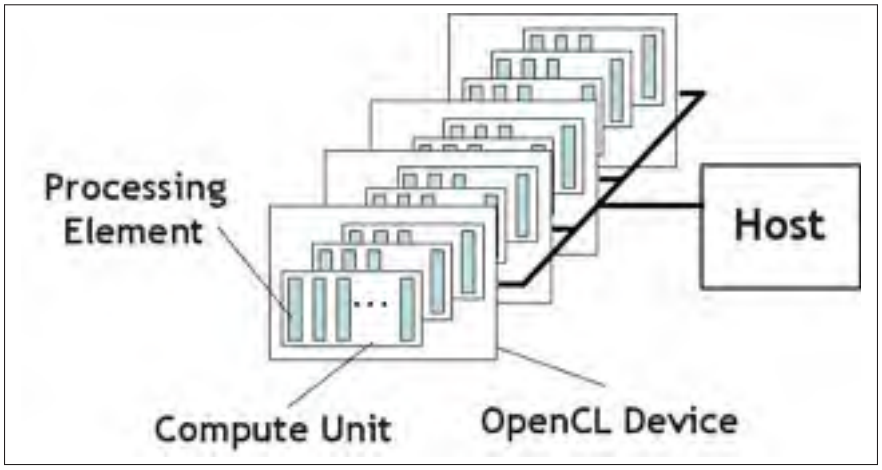


Figure 1.2 Structure de plateforme de calcul parallèle avec OpenCL
Tirée de OpenCL (2015)

OpenCL fait donc la distinction entre l'application tournant sur le processeur hôte (et qui va appeler l'API OpenCL) d'une part, et les noyaux qui sont programmés en OpenCL-C (et dont la vocation est d'être exécutés sur les périphériques) d'autre part. Les tâches à exécuter peuvent être créées dynamiquement via l'API OpenCL. Chaque tâche peut être représentée soit sous forme d'une instance unique, appelée tâche, soit sous forme d'une collection d'instances, appelées NDRange. La structure générale est représentée dans la figure 1.3. Les NDRanges peuvent être de 1, 2 ou 3 dimensions. Les NDRanges renferment un ensemble de groupes de travail. Chaque instance de kernel qui fait partie d'un groupe de travail est appelée unité de travail ou thread. Les unités de travail appartenant à un même groupe de travail peuvent partager des données et se synchroniser via des barrières.

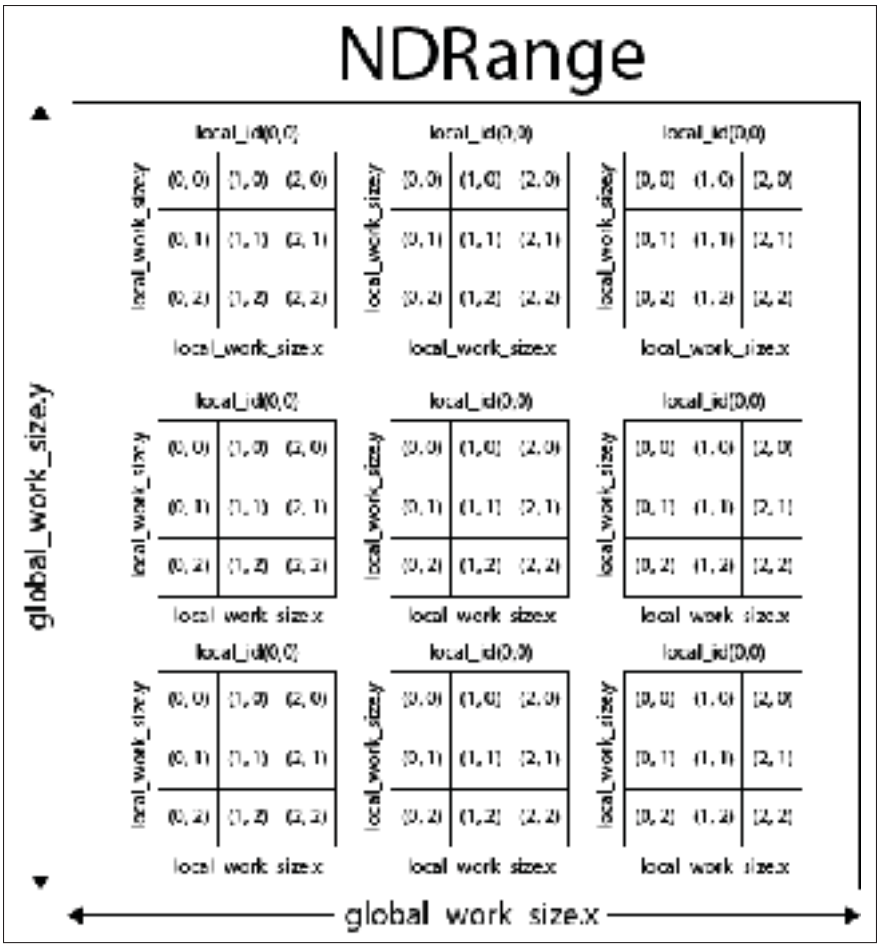


Figure 1.3 Disposition des threads dans une plateforme OpenCL
Tirée de OpenCL (2015)

1.4 Paradigmes du calcul parallèle

Pour accélérer le temps d'exécution des programmes informatiques, il existe deux solutions possibles :

- L'augmentation de la fréquence d'horloge de l'unité de calcul ce qui augmente le nombre d'opérations par unité de temps.
- La multiplication du nombre d'unités de calcul et l'exécution de plusieurs tâches en parallèle.

Pour la première solution, aucune expertise n'est requise pour le programmeur. Cependant pour la deuxième solution, il est indispensable de savoir comment exploiter les différentes unités de calcul pour augmenter la performance du système et permettre une exécution optimale des tâches en parallèle. L'augmentation de la fréquence d'horloge et la condensation des transistors dans les circuits des microprocesseurs engendrent une augmentation dans la consommation d'énergie d'une façon non linéaire (Lee et Gaudiot, 2003). Ainsi les systèmes de refroidissement deviennent inefficaces pour maintenir la stabilité du système. Cette limite des microprocesseurs qui est difficile à franchir a favorisé le terrain devant les systèmes parallèles pour accélérer la performance de calcul.

Il existe plusieurs classifications pour les architectures parallèles. En effet, proposée par (Flynn, 1972), cette taxonomie est considérée parmi les premières qui classifient les architectures et les programmes informatiques selon le flux d'instructions et le flux de données. On distingue alors quatre types d'architectures : SISD, SIMD, MISD, et MIMD.

Jusqu'à la fin des années 1990, l'architecture SISD (Single Instruction Single Data) était prédominante. On parle ici des ordinateurs offrant le calcul séquentiel et qui traitent une donnée à la fois.

Quant à l'architecture SIMD (Single Instruction Multiple Data), elle caractérise les systèmes parallèles qui exécutent une même instruction sur des données différentes. C'est le cas des systèmes multicœurs ou les processus vectoriels qui gèrent les applications multimédias ou scientifiques ayant des structures régulières (exemple les matrices).

Les architectures multiprocesseurs ou dotées de processeur multicœur sont plus polyvalentes et offrent pleinement le calcul parallèle. Ce sont des MIMD (Multiple Instructions, Multiple Data) qui exécutent plusieurs instructions sur des données différentes au même temps. Enfin l'architecture MISD (Multiple Instruction Single Data) consiste à exécuter plusieurs instructions sur un seul flux de données. Ce type d'architectures a été beaucoup plus rarement utilisé. Il semble néanmoins adapté à certains problèmes comme les réseaux de neurones et aux problèmes temps-réel.

Cependant, dans le cas réel certaines machines peuvent être hybrides selon cette classification qui est simple et donne une première approximation. Ainsi, cette approche de classification montre clairement deux types de parallélismes : le parallélisme par flot d'instructions (également nommé parallélisme de traitement ou de contrôle), et le parallélisme de données, où les mêmes opérations sont répétées sur des données différentes.

Il existe néanmoins une autre classification des systèmes parallèles basée sur le type de mémoire utilisé : les systèmes à mémoire partagée, les systèmes à mémoire distribuée et les systèmes hybrides.

1.4.1 Les systèmes à mémoire partagée

Dans ce type de systèmes, un même espace mémoire est partagé entre différentes unités de calculs parallèles. Ainsi des outils de synchronisation sont nécessaires pour empêcher l'écriture simultanée dans une zone mémoire pour assurer la cohérence des données. Parmi les outils de synchronisation de la programmation concurrente, on peut citer les sémaphores. Introduit par Edsger Dijkstra, un sémaphore est par définition une variable de type de donnée abstrait permettant la restriction d'accès à des ressources partagées ainsi que l'élimination de l'interblocage des processus parallèles.

1.4.2 Les systèmes à mémoire distribuée

Dans ce type de systèmes, chaque unité de calcul a son propre espace mémoire. Une organisation adéquate des données est alors nécessaire pour maintenir la cohérence du système. Dans une telle architecture, la communication inter processus est effectuée utilisant la communication réseau, et à travers des mécanismes de passage de messages.

1.4.3 Les systèmes hybrides

Dans un système hybride, on utilise les deux types de mémoires : partagées et distribuées. En pratique, cette topologie est la plus utilisée puisqu'elle combine les deux techniques. Les

différentes unités de calculs parallèles localisées dans différentes machines et ayant chacune sa propre mémoire sont connectées via une topologie réseau et exécutent d'une façon répartie les tâches en parallèle.

1.4.4 Les clusters

Parmi les architectures qui peuvent être classées dans l'un des deux derniers types de systèmes, on peut citer les clusters. D'après Tinetti (2000), un cluster est par définition un groupe d'unités de traitement reliées par une connexion réseau et qui peuvent être vues comme une seule unité exécutant des traitements en parallèle et destinés en général pour les calculs à haute performance. Il existe plusieurs types de clusters. On peut citer par exemple les clusters à haute disponibilité, les clusters MOSIX et enfin les clusters Beowulf.

Un cluster à haute disponibilité comme l'indique son nom possède la particularité d'avoir une tolérance zéro aux pannes. En cas de défaillance d'un noeud, les autres noeuds du cluster à haute disponibilité prennent en charge les fonctionnalités du nœud défaillant d'une manière transparente. Ce type de cluster est généralement utilisé pour les serveurs DNS, proxy ainsi que les serveurs Web.

Les clusters MOSIX utilisent une distribution open source du système d'exploitation Unix. Avec ce type de clusters, les utilisateurs peuvent exécuter plusieurs processus en permettant MOSIX de rechercher des ressources et d'attribuer automatiquement les processus aux différents nœuds, sans changer ni l'interface ni l'environnement d'exécution au niveau des noeuds. Aucune modification ni intégration de nouveaux modules n'est donc necessaire pour l'exécution des tâches au niveau des noeuds. Tout s'exécute d'une façon transparente par rapport à l'utilisateur (Barak et Shiloh, 1999).

Les clusters Beowulf quant à eux furent développés à la NASA en 1994. Le but de ce type d'architecture est de construire des unités de traitement parallèle interconnectées sous le système Linux tout en utilisant des composantes peu chères. Ce n'est qu'en 1995 que les premières réalisations concrètes de ce cluster ont vu le jour. Le système était composé de 16 PCs de type

486DX4@100MHz avec chacun deux interfaces Ethernet à 10Mb/s ainsi que des disques durs de capacité 250Mo chacun. Ce cluster Beowulf était utilisé pour accélérer la simulation de phénomènes physiques ainsi que l'acquisition de données dans le centre de la NASA. Aujourd'hui, les clusters Beowulf ont beaucoup évolué et ils sont classés dans les meilleurs 1000 supercalculateurs les plus puissants au monde. La taille de ce type de clusters peut aller aujourd'hui jusqu'à plusieurs dizaines de stations formées chacune de plusieurs multiprocesseurs et totalisant des Téra-octets de capacité de disque dur ainsi que de plusieurs dizaines de Giga-octets de RAM.

Les principaux standards les plus utilisés dans les architectures de clusters sont PVM (Parallel Virtual Machine) et MPI (Message Passing Interface). Le PVM est un standard pour le passage de messages entre des unités de calcul hétérogènes et interconnectées dans un réseau. Le PVM permet l'agrégation des noeuds d'un cluster en une seule machine virtuelle permettant ainsi d'augmenter la performance totale de calcul et augmenter ainsi le débit d'exécution des tâches lourdes.

MPI est aussi un standard répandu pour l'envoi et la gestion des taches parallèles dans les systèmes distribués. Cette interface a été conçue pour obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. L'avantage majeur de MPI par rapport aux plus vieilles bibliothèques de passage de messages est sa portabilité ainsi que la rapidité d'exécution.

1.4.5 Efficacité des systèmes parallèles

Intuitivement, l'accélération d'un programme avec le calcul parallèle doit être linéaire. Si on double par exemple le nombre des unités de calcul, la moitié du temps d'exécution doit être obtenu. Cependant, une minorité de programmes possède ce type de performance. En effet, d'après la loi d'Amdahl établie en 1960, la partie du programme non parallèle limite l'accélération du programme. On effet, d'après cette loi l'accélération peut être mesurée par la formule suivante :

$$S(n) = \frac{1}{1 - p + \frac{p}{n}} \tag{1.1}$$

où

- S(n) : L'accélération en fonction du nombre de processeurs n
- n : Le nombre de processeurs en parallèle
- p : La fraction du temps d'exécution de la tâche concernée par l'amélioration

Comme c'est illustré dans la figure 1.4, si nous augmentons le nombre de processeurs en parallèle l'accélération atteindra toujours un seuil fixe qui dépend de la portion séquentielle de la tâche. Dans la pratique, la loi d'Amdahl est utilisée pour fixer une certaine limite à la performance des architectures parallèles ou à l'optimisation de la programmation pour la résolution d'un problème donné. Il existe une autre loi qui s'intéresse à l'efficacité du calcul parallèle appelée la loi de Gustafson. Cette loi affirme que si on augmente le nombre de processeurs en parallèle, une quantité plus importante de données peut être traitée en un temps équivalent. La loi d'Amdahl, quant à elle, fixe une limite d'efficacité de traitement à quantité de données égale.

1.4.6 Avantages de l'utilisation des systèmes parallèles et distribués

Nombreuses sont les motivations qui proviennent de l'utilisation des systèmes parallèles et distribués. Les points ci-dessous reflètent quelques-unes.

- **Augmentation de la performance :** Une des raisons classiques pour l'utilisation des architectures parallèles est la demande continue pour l'amélioration des performances des systèmes informatiques en accélérant le traitement et diminuant le temps de latence. Le développement d'architectures informatiques de plus en plus complexe et sophistiqué requiert un plus haut niveau de calculs avancés ce qui rend l'utilisation des architectures parallèles et distribuées inévitable.

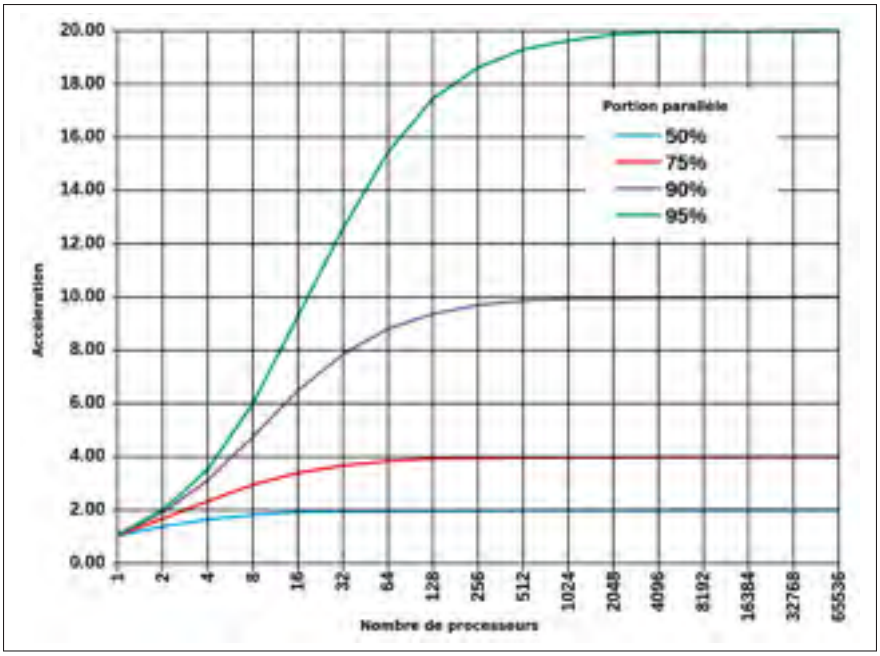


Figure 1.4 La loi d'Amdahl

- **Compromis entre performance et prix :** Au niveau du marché, les architectures parallèles sont beaucoup moins chères que les architectures séquentielles ayant la même performance (Padua, 2011). En effet, concevoir un nouveau processeur coute très cher et il est également difficile à l'intégrer dans la majorité des systèmes informatiques (coût de migration supplémentaire).
- **Raisons techniques :** Les limitations physiques pour l'augmentation du nombre des transistors au niveau des processeurs et le problème de dissipation de chaleur favorisent la migration vers les architectures parallèles. Le coût réduits des processeurs et des composants hardwares favorisent leur utilisation à grande échelle au sein du même système. Une autre restriction pour l'utilisation des architectures séquentielle est la dégradation de la performance à cause du temps de latence causé par l'accès à la mémoire. L'exploitation de composants offrants le calcul parallèle favorise l'utilisation des mémoires caches et font en sorte que les données soient traitées près des emplacements où ils sont générés (c'est le cas par exemple des GPU et des coprocesseurs) dans le but de diminuer le coût de transfert des données.

- **Partage de ressources :** De nos jours de plus en plus de ressources sont disponibles et interconnectées surtout avec le développement des réseaux informatiques (tablettes, téléphones mobiles, ordinateurs, cartes embarquées, serveurs, etc.). Les systèmes distribués offrent la possibilité de partage des ressources ainsi que la répartition des tâches pour accélérer le traitement et réduire les coûts de traitement (accès au plus de mémoires, plus de processeurs, etc.).
- **Disponibilité :** Les systèmes parallèles et distribués possèdent plusieurs composantes du même type. Si l'une de ces composantes n'est plus disponible, on peut en général avoir recours aux autres unités de calcul pour continuer le traitement. La haute disponibilité constitue un critère majeur pour les grands systèmes répartis et critiques comme le cas par exemple des bases de données pour les applications commerciales.
- **Capacité à monter en charge :** Les systèmes parallèles et distribués constituent de bons candidats ayant la capacité à accroître leur performance de calcul sous une charge accrue quand des ressources matérielles sont ajoutées.

1.5 Les techniques de détection de malwares et le besoin d'optimisation et du parallélisme

Le problème de sécurité des téléphones intelligents diffère de celle des ordinateurs personnels. Ceci est dû aux différences architecturales de ces appareils et la multitude d'infrastructures spécialisées sur laquelle reposent ces téléphones. Cette diversité constitue néanmoins un défi majeur pour les concepteurs et accroît la complexité ainsi que le temps nécessaire à l'analyse et au développement de sécurité pour ces appareils. Il existe plusieurs aspects qui distinguent la sécurité des appareils mobiles de celle qui est conventionnelle. On peut résumer ces aspects sur les points suivants : une mémoire et une capacité de calcul réduite, une capacité pour le calcul parallèle en évolution, une mobilité, une forte connectivité et une forte personnalisation.

- ***Mémoire et capacité de calcul réduite :*** Le budget mémoire limité ainsi que la capacité de calcul réduite des téléphones mobiles étaient toujours une contrainte majeure pour l'utilisateur. Ainsi, au cours des dernières années, les firmes de la téléphonie mobiles ne cessent

de travailler pour augmenter les performances de leurs Smartphones. Ceci est illustré par la courbe de la figure 1.5 reflétant l'évolution de la mémoire vive des téléphones intelligents au cours de dernières années. Pour mieux saisir l'importance de cet aspect, prenons l'exemple du système d'exploitation Android opérant sur un téléphone Samsung Galaxy S. Ce téléphone possède 512MB de mémoire vive avec 36% d'espace mémoire consommé sans lancer aucune autre application. En outre, la mémoire requise pour chaque application dépend de sa complexité (soit par exemple 27MB pour Skype et 35MB pour Google Maps pour le même système).

- ***Mobilité :*** Les Smartphones sont conçus pour être mobiles. Dans la plupart du temps, on ne les garde pas dans un endroit sécurisé et peuvent donc être facilement volés ou manipulés.
- ***Forte connectivité :*** Multiples sont les façons avec lesquelles un appareil mobile peut se connecter à un réseau internet. Ceci augmentera le risque d'attaque et rend le téléphone plus vulnérable aux menaces de sécurité.
- ***Forte personnalisation :*** Les téléphones mobiles figurent parmi les affaires personnelles de l'usager. Ces appareils ne sont jamais loin de leur propriétaire et comprennent donc les informations personnelles de ce dernier comme le carnet d'adresses, la liste des messages, les photos, etc.

Tous ces éléments rendent la sécurité pour les téléphones mobiles plus délicate. En regroupant tous ces aspects, on peut voir la raison pour laquelle la sécurité des appareils mobiles est plus complexe que celle des ordinateurs normaux. Prenons l'exemple de la mobilité. Cette caractéristique des Smartphones augmente le risque de vol des données personnelles stockées. En effet, voler un téléphone portable est beaucoup plus simple que de pénétrer dans un ordinateur personnel. En outre, une forte connectivité facilite la violation de la vie privée de l'utilisateur (un appareil mobile est la plupart du temps auprès de son propriétaire, donc si nous arrivons à localiser le Smartphone, nous aurons l'emplacement du propriétaire). Enfin, les fonctionnalités réduites ainsi que les performances limitées des téléphones mobiles peuvent être une source d'attaques dites par déni de service. Ceci a pour but de rendre le téléphone mobile inutilisable.

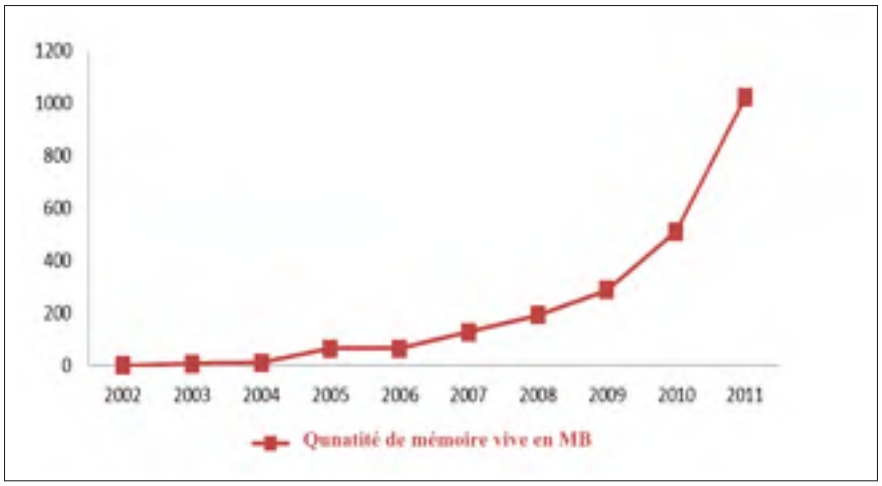


Figure 1.5 Évolution de la mémoire vive pour les Smartphones
Tirée de Amamra *et al.* (2012b)

Tous ces aspects, et bien d'autres engendrent de nouvelles implications, comme le fait de rendre l'audit de la sécurité sur les appareils mobiles plus complexe. Ainsi, le besoin d'accélération de la détection d'attaques pour assurer un plus haut niveau de sécurité devient de plus en plus important surtout avec les capacités de calcul parallèle émergentes offertes par les GPU mobiles (comme nous l'avons détaillé dans la section précédente). Plusieurs sont les avantages de l'utilisation de la GPU mobile à des fins de sécurité. En effet, cet élément offre une unité de calcul complémentaire à la CPU. Ensuite, les GPUs peuvent être exploitées quand ce dernier possède une charge de traitement importante. Enfin, les GPUs ne sont pas toujours totalement utilisées quand le téléphone mobile est actif. Nous pouvons alors en tirer profit à travers l'envoi des charges lourdes de la CPU pour assurer une utilisation optimale des ressources disponibles.

1.5.1 Méthodes de détection de malwares dans les Smartphones

Un détecteur de malwares est un système responsable de déterminer si un programme possède un comportement malicieux ou pas. Diverses sont les approches adoptées pour la détection de malware dans les téléphones mobiles. La figure 1.6 illustre une classification possible d'approches de détection de malwares. On peut classifier ces techniques sur deux grands volets : les techniques basées sur les signatures de malwares et celles basées sur la détection d'anomalies.

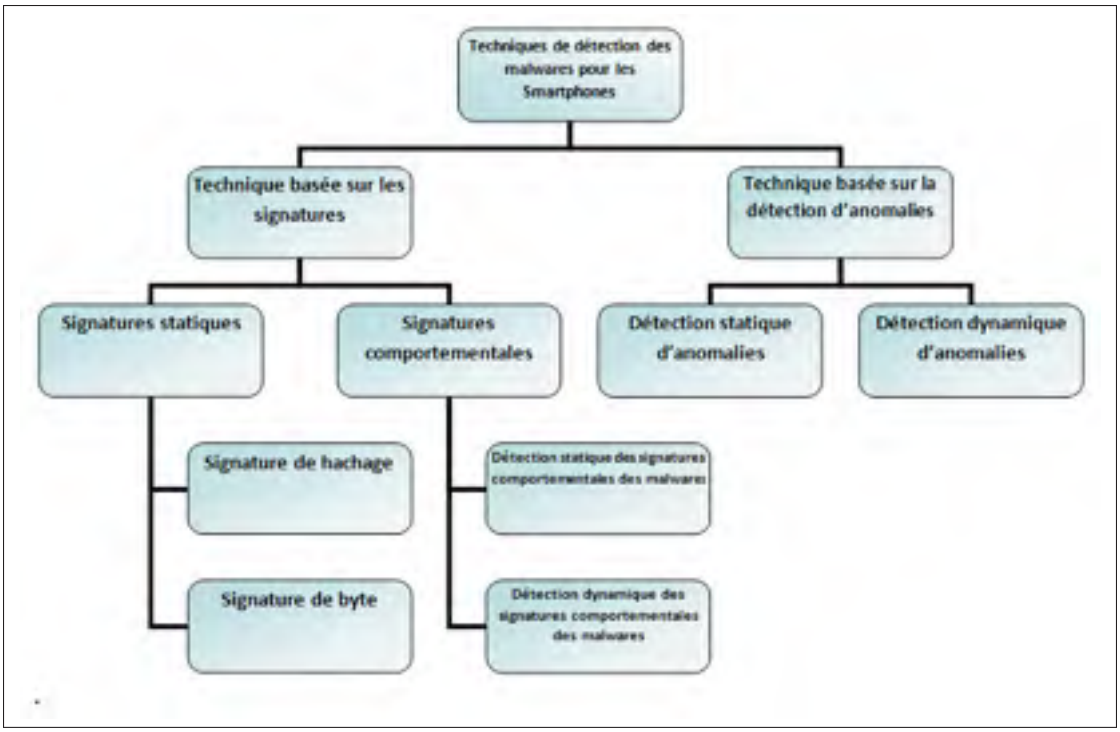


Figure 1.6 Classification des méthodes de détection de malwares sur les téléphones mobiles
Tirée de Amamra *et al.* (2012a)

1.5.1.1 Détection d'anomalies

Une anomalie est par définition " *tout ce qui s'écarte de la norme, de la régularité, de la règle*" **?**. Cette approche est basée sur deux phases : phase d'apprentissage et phase de détection. Au cours de la phase d'apprentissage, on construit la base de données du comportement normal des applications. Ensuite, au cours de la phase de détection, chaque déviation de ce profil d'exécution est considérée comme anomalie.

La détection d'anomalie peut être classifiée en deux catégories : la détection statique et la détection dynamique. Plusieurs sont les points de références qu'on peut utiliser dans la détection statique d'anomalies. En effet, pour déterminer les inconsistances du programme avec le comportement normal on peut se baser sur les informations statiques du code comme la syntaxe, la structure du programme, les données et les chemins de contrôles. Parmi les points forts de la détection statique d'anomalies, c'est qu'elle découvre les vulnérabilités du code ainsi

que l'existence de malwares avant leur exécution. Cependant cette technique est complexe et requière des ressources de calcul importantes. De plus la construction des modèles de comportements normaux reste problématique.

Concernant la détection dynamique d'anomalies, c'est dans la phase d'apprentissage qu'on collecte les traces d'exécution des applications et on construit par la suite la base de données de comportement normal des programmes. Au cours de la phase de détection, on fait le monitorage du comportement des applications en cours d'exécution et on détecte s'il y a eu la présence d'anomalies. Cette technique est proche de la détection statique d'anomalies. La seule différence entre ces deux techniques réside dans la collecte dynamique des traces d'exécution qui peut être des fichiers log, des appels systèmes, des appels API, etc. Cette approche permet de détecter de nouveaux malwares ainsi que les attaques dites "jour-zéro". Ce type d'attaque est un virus ou un exploit qui profite d'une faille nouvellement découverte dans un programme ou un système d'exploitation avant que les développeurs n'aient mis à disposition un correctif ou avant même qu'ils n'aient pris conscience de l'existence d'une telle faille. Une attaque jour zéro sur l'ordinateur de bureau d'un utilisateur particulier peut affecter la productivité de ce dernier. Déclenchées sur la dorsale de réseaux mondiaux, les attaques jour zéro peuvent littéralement interrompre toutes les activités et être directement responsables de pertes de temps, d'argent et de productivité.

1.5.1.2 Détection des signatures de malwares

Cette technique identifie les virus à travers l'élaboration des signatures spécifiques à chaque malware. Ces signatures sont stockées dans une base de données qui sera consultée à chaque fois où on veut détecter s'il y a eu une attaque au système qu'on veut protéger. Plus la base de données des signatures est mise à jour, plus le détecteur des malwares est efficace. La signature de malware sur laquelle repose cette approche de détection d'intrusion peut être statique ou comportementale.

1.5.1.2.1 Signatures statiques de malwares

C'est la technique la plus utilisée dans la plupart des antivirus qui se trouvent sur le marché comme MacFee, Kespersky, Norton, etc. Ils analysent la mémoire vive du téléphone ainsi que les fichiers enregistrés dans le but de trouver des patterns qui coïncident avec l'une des signatures statiques des malwares, stockées dans la base de données. Les types de signatures les plus utilisées dans cette technique sont les signatures de byte et les signatures basées sur les fonctions de hachage. La contrainte majeure de cette technique réside dans le fait que si on ajoute des lignes de codes qui n'affectent pas le fonctionnement du malware on obtient une nouvelle signature. La détection de malwares basée sur les signatures statiques ne permet pas de détecter de nouvelles attaques ainsi que leurs variantes. Cependant cette technique ne requière pas des capacités de calculs énormes et elle est donc plus utilisée dans les antivirus qu'on trouve sur le marché.

1.5.1.2.2 Signatures comportementales de malwares

Les signatures comportementales sont des métastructures complexes qui nous renseignent sur le comportement général du logiciel malveillant. L'analyse comportementale consiste à analyser les actions qu'un programme effectue pour, s'il est nécessaire, en déduire qu'il a un comportement douteux et qu'il vaut mieux prévenir l'utilisateur d'un potentiel risque encouru par l'utilisation de ce fichier. Les actions potentiellement dangereuses peuvent être par exemple la suppression et/ou modification de fichiers sensibles, l'accès à une zone mémoire protégée, l'accès en lecture et/ou écriture à un fichier exécutable, etc. Un des problèmes résultants de cette méthode est le fait que les actions potentiellement dangereuses sont exécutées avant qu'un quelconque avertissement n'est pu être transmis à l'utilisateur.

Nombreuses sont les données qui peuvent être exploitées par un tel type de détection. On peut citer par exemple l'étude des appels systèmes émis par les processus en cours d'exécution, les appels API, les permissions que possèdent les applications, l'analyse des graphes d'appels des fonctions, etc.

1.6 Algorithmes de Pattern matching

Les anti-malwares reposent majoritairement sur un mécanisme de comparaison de motifs. La performance du système de détection est déterminée non seulement par sa base de données, mais aussi par le degré de fiabilité d'algorithme de correspondance de patrons. Vu que nous sommes dans un contexte de téléphone mobile, la contrainte de la mémoire ainsi que le processeur jouent un rôle primordial dans le choix de l'algorithme à utiliser. On peut représenter le problème plus formellement : étant donné un alphabet $\sum$, un ensemble de mots P = { p_1,p_2,...,p_m } et un texte T = { t_1,t_2,...,t_n } , on veut trouver toutes les occurrences exactes de tous les mots de P dans T. Les occurrences peuvent se chevaucher. Le problème de recherche de patterns dans le domaine de sécurité requiert la prise en considération de plusieurs facteurs :

a. ***La recherche de plusieurs patterns à la fois*** : il existe deux types de données qu'on manipule avec ces algorithmes qui sont les patterns et le texte à chercher. Les patterns sont déjà statiques et prédéfinis. Cependant le texte en entrée est dynamique et change à chaque fois où l'on reçoit un paquet. Le but de ces algorithmes est donc de détecter la présence des patterns dans le texte reçu.

b. ***La taille d'un pattern*** : La performance de certains algorithmes ainsi que leur complexité dépend de la taille des patterns utilisés.

c. ***La taille de l'ensemble des patterns*** : Plus le nombre de motifs est grand plus la mémoire qu'on va exploiter est grande. Ceci affectera la performance du système.

d. ***La taille de l'alphabet*** : Elle a un impact significatif sur la rapidité de l'algorithme. Il faut bien choisir l'alphabet pour avoir de bons résultats

e. ***La taille du texte sur lequel on appliquera la recherche*** : Certains algorithmes ont une complexité linéaire par rapport au texte d'entrée. On peut citer à titre d'exemple l'algorithme Aho-Corasick qu'on va détailler par la suite.

f. ***La fréquence de recherche*** : Elle dépend de la bande passante du réseau, le quantité de trafic ainsi que la taille des paquets reçus. Une haute fréquence de recherche peut consommer plus de mémoire et affecter le fonctionnement général du téléphone. Une

basse fréquence de recherche peut ne pas être efficace et induire à un retard de détection d'intrusion.

En ce qui suit, nous allons détailler les principaux algorithmes de correspondance de patrons et discuter leurs performances.

1.6.1 L'algorithme de Aho-Corasick

Élaboré par Alfred Aho et Margaret Corasick, l'algorithme de Aho-Corasick (Aho et Corasick, 1975) est considéré comme référence dans le domaine de correspondance de patrons. Cet algorithme repose sur la construction d'un automate fini déterministe à partir de la liste des motifs. Ce type d'automates est constitué d'un ensemble d'états S et d'une fonction de transition g tel que pour tout état s et pour tout symbole a de l'alphabet $\sum$, $g(s,a)$ est un état de S. En d'autres termes, on ne peut effectuer qu'une seule transition pour chaque symbole en entrée. La construction de l'automate se fait dans une phase appelée prétraitement. Quant au processus d'analyse, il est élaboré dans la phase de traitement.

1.6.1.1 Phase de prétraitement

La machine de correspondance de patrons est constituée d'un ensemble d'états. Chaque état est identifié par un numéro unique. Cet algorithme repose sur trois axes principaux qui sont la matrice de transitions , les transitions d'échec et les états de sortie.

- ***matrice de transitions***

  En littérature et dans le contexte de cet algorithme, la matrice de transitions est représentée par une fonction appelée "GoTo". Elle prend comme paramètres l'état actuel et l'alphabet en entrée puis donne comme résultat l'état de destination ou échec. Prenant par exemple la liste des patterns suivante : "he", "she", "his", "hers". La figure 1.7 illustre l'introduction des motifs dans l'automate.

- ***Les états de sortie***

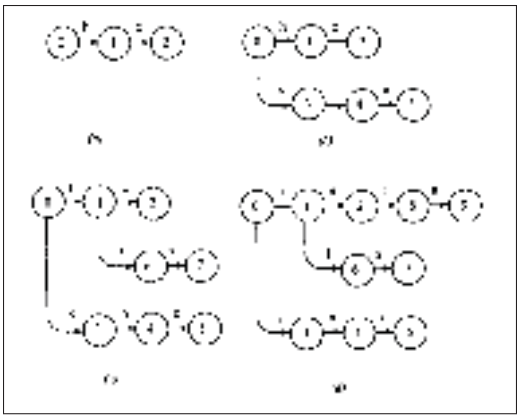


Figure 1.7 Introduction des patrons dans l'automate
Tirée de Aho et Corasick (1975)

A la fin de l'ajout de chaque motif on marque l'état comme étant un état de sortie en indiquant le numéro du patron correspondant.

Tableau 1.1 Tableau des états de sortie

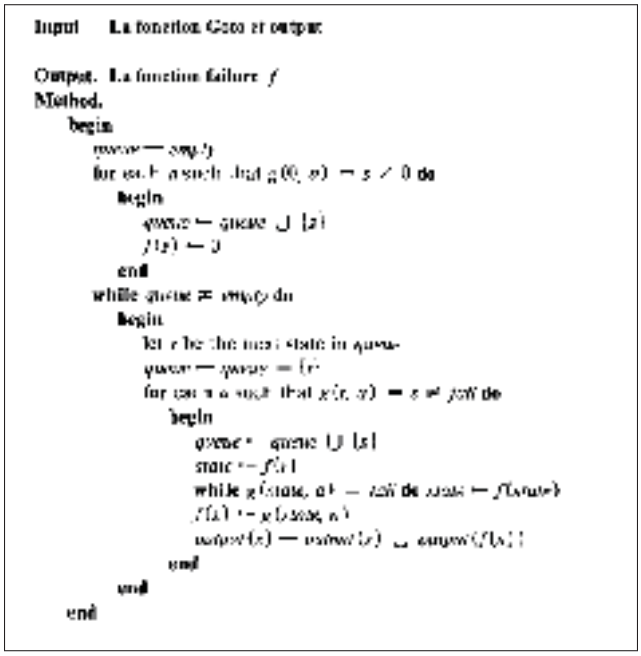
État	0	1	2	3	4	5	6	7	8	9
Sortie	0	0	1	0	0	2	0	3	0	4

- ***Les transitions d'échec***

  Au cours de la phase de prétraitement, on élabore aussi les transitions d'échec de l'automate. En effet, nous commençons par la détermination de ces transitions pour les états de profondeur un, puis pour les états de profondeurs deux, et ainsi de suite jusqu'à ce qu'on parcourt tous les états sauf l'état initial.

1.6.1.2 Phase de traitement

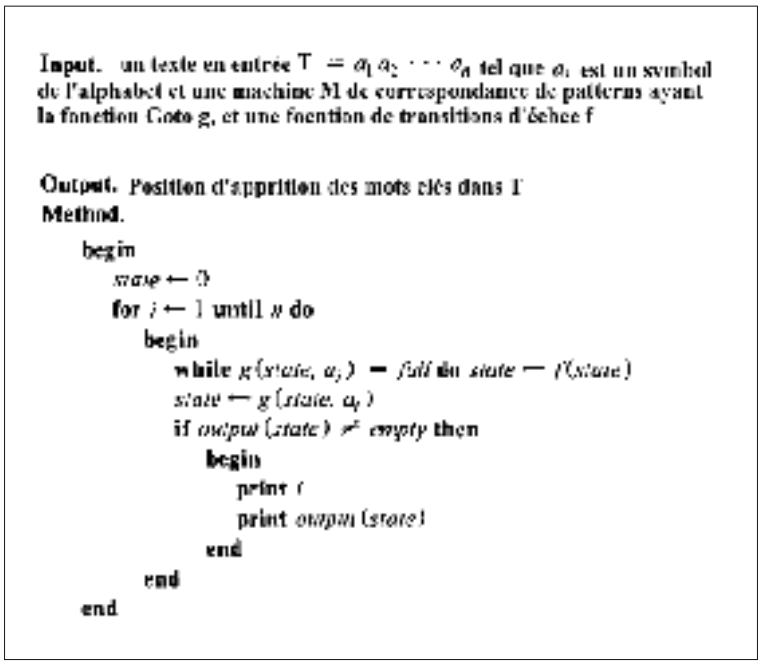
C'est à cette étape qu'on déclenche le processus d'analyse pour déterminer la correspondance entre les motifs P et le texte en entrée T. L'algorithme responsable de ce traitement est détaillé dans la figure 1.9.



Input La fonction Goto et output

Output. La fonction failure f
Method.
begin
$queue \leftarrow empty$
for $a \in A$ such that $g(0, a) = s \neq 0$ do
begin
$queue \leftarrow queue \cup \{s\}$
$f(s) \leftarrow 0$
end
while $queue \neq empty$ do
begin
let r be the next state in $queue$
$queue \leftarrow queue - \{r\}$
for $a \in A$ such that $g(r, a) = s \neq fail$ do
begin
$queue \leftarrow queue \cup \{s\}$
$state \leftarrow f(r)$
while $g(state, a) = fail$ do $state \leftarrow f(state)$
$f(s) \leftarrow g(state, a)$
$output(s) \leftarrow output(s) \cup output(f(s))$
end
end
end

Figure 1.8 Algorithme de détermination des transitions d'échec
Tirée de Aho et Corasick (1975)



Input. un texte en entrée $T = a_1 a_2 \cdots a_n$ tel que a_i est un symbol de l'alphabet et une machine M de correspondance de patterns ayant la fonction Goto g, et une fonction de transitions d'échec f

Output. Position d'apparition des mots clés dans T
Method.

```
begin
    state ← 0
    for i ← 1 until n do
        begin
            while g(state, a_i) = fail do state ← f(state)
            state ← g(state, a_i)
            if output(state) ≠ empty then
                begin
                    print i
                    print output(state)
                end
        end
end
```

Figure 1.9 Algorithme de détection
d'apparition des patrons dans le texte en entrée
Tirée de Aho et Corasick (1975)

1.6.2 Autres algorithmes de pattern matching

Multiples sont les algorithmes utilisés pour le pattern matching. Parmi ces techniques on peut citer l'algorithme SBOM (Allauzen *et al.*, 2001) qui repose sur la construction d'un automate à partir de patrons organisés à l'envers. L'analyse de la chaîne de caractères en entrée se fait alors de droite vers la gauche. On peut citer également l'algorithme de Wu-Manber (Wu *et al.*, 1994) qui repose sur l'analyse de tout un bloc de caractères, au lieu de les prendre un par un. Cet algorithme manipule des techniques de décalage et de table de hachage pour établir la correspondance des patrons avec le texte en entrée. Enfin, on peut encore citer l'algorithme de Comment-Walter (Commentz-Walter, 1979) qui ressemble à celui de Aho-Corasik et qui utilise des techniques de saut de caractères. Une étude comparative des ces différents algorithmes à été effectuée par Amamra *et al.* (2012b). Cette étude s'est élaborée en se basant sur des patrons représentant les signatures statiques des malwares. Le tableau illustré dans la figure 1.10 résume la consommation de la mémoire de ces algorithmes implémentés sur un téléphone Android de type HTC.

Les résultats représentés ci-dessous reflètent la consommation importante de la mémoire des algorithmes cités ci-dessus, ainsi que l'inefficacité de certains, à partir d'un certain seuil de données. En effet, avec l'algorithme WM nous avons un dépassement de la mémoire à partir de mille quatre cents signatures. Quant au Comment-Walter (CW), ce n'est qu'à partir d'un total de mille cinq cents signatures, que l'algorithme devient inefficace. La consommation de la mémoire de ces algorithmes reste importante, ce qui augmente le besoin d'optimisation de la mémoire dans le domaine de correspondance des patrons.

1.7 Algorithmes de cryptographie

La cryptographie est par définition l'art de chiffrer ou de coder les messages. Elle est devenue aujourd'hui une science à part entière vu son importance et son vaste domaine d'application. Au croisement des mathématiques, de l'informatique, et parfois même de la physique, elle offre ce dont les civilisations ont toujours besoin depuis leur existence : le maintien du secret.

Signature de malwares	Mémoire requise (MB)		
	CW	WM	SBOM
500	20.87	22.70	14.08
600	26.34	28.60	18.67
700	32.83	34.90	22.54
800	35.92	39.26	25.15
900	40.87	43.58	32.65
1000	45.09	49.20	34.22
1100	48.70	53.75	39.07
1200	51.98	58.09	43.66
1300	57.67	64.12	46.18
1400	63.59	Dépassement de la mémoire	51.07
1500	Dépassement de la mémoire	Dépassement de la mémoire	55.98
1600	Dépassement de la mémoire	Dépassement de la mémoire	61.04

Figure 1.10 Consommation de la mémoire de quelques algorithmes de correspondance de patrons
Tirée de Amamra *et al.* (2012b)

La cryptographie a comme but l'étude de solutions permettant d'assurer trois services : l'intégrité, l'authenticité et la confidentialité des systèmes d'information et de communication. Ce domaine est conjoint également à l'ensemble des outils informatiques qui doivent résister à des éléments ayant pour but de nuire aux services offerts par un tel mécanisme. C'est à l'aide de la cryptanalyse qu'on recherche des failles dans les mécanismes proposés.

Dans cette section nous adopterons la terminologie suivante :

- Le message clair : est le message d'origine.
- Le chiffrement : est la transformation effectuée sur le texte clair.
- Le texte chiffré ou cryptogramme : est le message transformé par un algorithme de chiffrement.
- Le déchiffrement : est la transformation de reconstitution sur un texte.

On peut classifier les algorithmes de cryptographie en deux types :

- La cryptographie symétrique : on utilise la même clé pour chiffrer et déchiffrer les messages.
- La cryptographie asymétrique : on utilise des clés différentes pour chiffrer et déchiffrer les messages.

1.7.1 Algorithmes de cryptographie symétriques

La cryptographie symétrique ou encore appelée cryptographie à clé secrète est la plus ancienne historiquement. Ce premier type de cryptographie est répandu grâce aux performances remarquables qu'il offre. Comme c'est illustré dans la figure 1.11, la cryptographie symétrique requiert au moins deux éléments qui partagent la connaissance de la même clé secrète. Cette même clé va être utilisée pour le chiffrement et le déchiffrement des messages. La cryptographie symétrique se base principalement sur les fonctions booléennes ainsi que les statistiques.

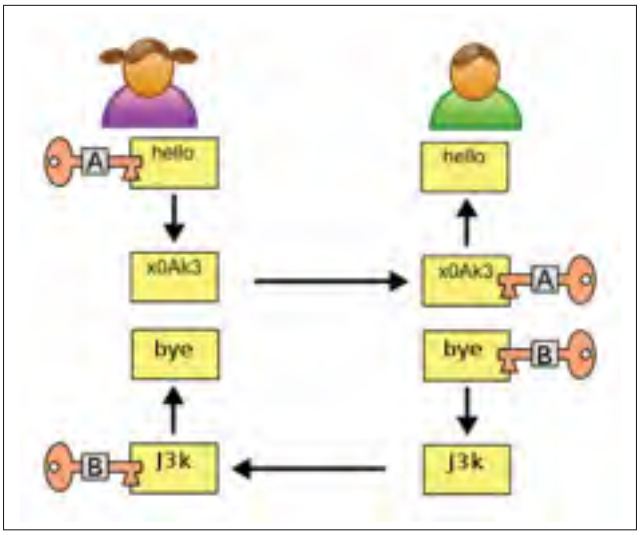


Figure 1.11 Cryptographie symétrique

En ce qui concerne les algorithmes de chiffrement symétrique, nous distinguons deux types d'algorithmes :

- chiffrement par bloc : le message clair est divisé en blocs relativement grands (128 bits par exemple) sur lesquels on effectue des opérations bien définies.

- chiffrement par flot : le message clair est vu comme un flot de bits ou d'octets, et il est combiné avec un autre flot généré d'une façon pseudo-aléatoire.

1.7.1.1 Le chiffrement par bloc

Un algorithme de chiffrement par bloc pour la cryptographie symétrique est généralement basé sur des modèles itératifs. On découpe le message clair à chiffrer en des blocs qui sont généralement de taille fixe. Une fonction d'itération est généralement utilisée qui prend en entrée la clé secrète et le message en clair. Pour chaque itération différentes clés déduites de la clé secrète initiale sont appliquées sur les blocs. Parmi les algorithmes de chiffrement symétrique par bloc on peut citer DES, AES et Blowfish.

1.7.1.1.1 DES

Développé par IBM en 1974, l'algorithme de chiffrement symétrique DES est devenu sous peu un standard en 1977. Cet algorithme utilise une clé de 56 bits ce qui le rend vulnérable aux attaques de cryptanalyse en un temps raisonnable (l'espace de clés est trop petit). L'algorithme consiste à exécuter une suite de combinaisons, de substitutions ainsi que des permutations entre le texte en clair et la clé de chiffrement. Pour augmenter le niveau de robustesse de l'algorithme DES, plusieurs variantes ont été développées. Le triple DES est l'une de ces variantes. Cette variante consiste à appliquer trois chiffrements DES en utilisant deux clés différentes de longueur égale à 56 bits chacune. Le triple DES permet d'augmenter certainement le niveau de la sécurité de l'algorithme DES. Cependant, ce processus a l'inconvénient majeur d'être plus lent et de demander plus de ressources pour le processus de chiffrement et de déchiffrement des messages.

1.7.1.1.2 AES

Annoncé par la NIST (National Institue of Standards and Technology) en 1997, l'algorithme AES est devenu un standard de cryptographie symétrique qui remplace le DES vu la faiblesse de ce dernier face aux attaques actuelles. En effet, la longueur d'une clé DES est égale à 56 bits si nous faisons abstraction des 8 bits de contrôle de parité. Cela signifie qu'il y a 2^{56} (soit à peu près 7.2 x 10^{16}) clés différentes possibles. Les clés de l'algorithme de cryptographie AES sont de longueur 128 bits. Ceci nous donne un nombre d'ordre 10^{21} fois plus grand de possibilités de clés pour l'AES que de clés à 56 bits pour l'algorithme DES. Si on suppose que nous pouvons construire un outil qui pourrait casser une clé DES en une seconde, ceci permet de faire l'hypothèse que cette machine peut calculer 2^{55} clés par seconde. Ainsi, nous pouvons conclure que pour casser une clé AES il faudrait encore 149 mille milliards d'années de traitement, ce qui construit la robustesse de tel algorithme de cryptographie symétrique.

Le principe de fonctionnement de l'AES est illustré dans la figure 1.12. En effet :

- BYTE_SUB : ou *Byte Substitution* représente une fonction non-linéaire appliquée d'une façon indépendante sur chaque bloc de données à partir d'une table appelée table de substitution.
- SHIFT_ROW : représente une fonction appliquant une suite de décalages. Cette fonction prend l'entrée en quatre segments de quatre octets et effectue des décalages vers la gauche de 0, 1, 2 et 3 octets pour les segments 1, 2, 3 et 4 respectivement.
- MIX_COL : est une fonction qui applique une suite de transformations d'octets à travers l'utilisation de produit matriciel.
- Ki : représente la ième sous-clé obtenue à partir de la clé secrète principale K.

  Quant au déchiffrement, il suffit d'appliquer les opérations précédentes à l'inverse.

L'algorithme AES construit particulièrement un bon candidat pour les implémentations dans les systèmes embarqués vu sa robustesse et la facilité d'implémentation d'une part, et la capacité de mémoire réduite qui est requise pour un tel système d'autre part. C'est sans doute

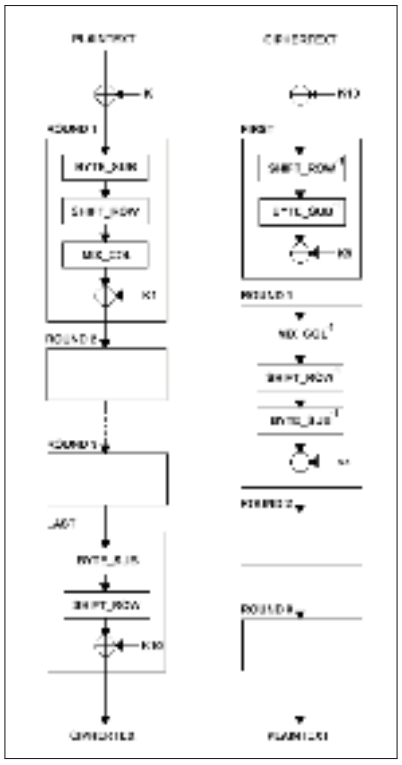


Figure 1.12 Schéma de fonctionnement de AES
Tirée de Miller *et al.* (2009)

ces éléments qui ont poussé le monde de la 3G (3ème génération du réseau mobile) à intégrer cette technique de cryptographie dans le mécanisme d'authentification des systèmes de télécommunication.

1.7.1.2 Le chiffrement par flot

On peut définir les algorithmes de chiffrement par flux ou par flot comme étant des algorithmes de chiffrement par blocs, où le bloc a une dimension unitaire (1 bit, 1 octet, etc.). Ce type de chiffrement possède plusieurs avantages :

- On peut changer la méthode de chiffrement au niveau de chaque symbole traité. Le chiffrement du texte en clair sera alors plus rapide.
- Utile dans les environnements où la fréquence d'erreurs est grande. Avec une telle technique de chiffrement, les erreurs de diffusions ne sont pas propagées.

- Utile dans les environnements où il est impossible de traiter un grand nombre d'informations (exemple mémoire tampon limitée).

Les techniques de chiffrement par flot appliquent des transformations simples selon une chaine de clés. Cette dernière peut être définie comme étant une séquence de bits utilisée en tant que clé qui peut être générée aléatoirement. Si nous travaillons avec une chaine de clés utilisée une seule fois et choisie aléatoirement, le message chiffré sera excessivement sécuritaire. Les algorithmes de chiffrement par flux sont beaucoup moins nombreux que les algorithmes de chiffrements par blocs. Cependant, ils sont de plus en plus répandus avec la circulation excessive des données sur le web. C'est dans ce domaine des logiciels que les algorithmes de chiffrement par flux ont toute leur importance.

Parmi les algorithmes de chiffrement par flot on peut citer :

- A5/1 qui est un algorithme utilisé dans les réseaux GSM pour le chiffrement des communications par radio effectuées entre le téléphone mobile et l'antenne-relais ;
- RC4 qui est un algorithme de chiffrement par flot conçu en 1987 par Ronald Rivest. Cet algorithme est beaucoup utilisé dans les protocoles WEP du Wi-Fi ;
- Py un algorithme récent de Eli Biham ;
- E0 qui est un algorithme de chiffrement par flot utilisé dans les communications Bluetooth.

1.8 Conclusion

Dans cette section nous avons introduit les différentes notions de base de notre travail de recherche. Dans le chapitre suivant, nous allons décrire une revue de littérature concernant principalement le calcul parallèle au service de la sécurité (les algorithmes de pattern matching et de la cryptographie parallèles), les techniques de détection de malwares mobiles utilisant les signatures ainsi que les différentes techniques de compactage de données puisque nous ciblons des plateformes à ressources limitées.

CHAPITRE 2

REVUE DE LA LITTÉRATURE

2.1 Introduction

Ce chapitre présente une revue de littérature dans les principaux axes de notre travail de recherche. Cette revue va permettre de mieux comprendre le domaine et de cibler nos contributions. En effet, dans une première section, nous allons étudier les techniques de détection de malwares mobiles qui sont basées sur l'utilisation des signatures malicieuses. Puis, nous allons décrire quelques techniques d'accélération d'algorithmes de correspondance de patrons avec le calcul parallèle. Ensuite, nous allons aborder dans une troisième section les techniques de compactage des structures d'automates vu les ressources limitées de l'environnement que nous ciblons. Nous allons traiter également dans une quatrième section quelques travaux sur la cryptographie parallèle. Enfin nous allons décrire l'utilisation des architectures des clusters qui existent dans la littérature et qui sont au service de la sécurité.

2.2 Techniques de détection de malwares mobiles basées les signatures

La sécurité mobile des devenue l'un des enjeux les plus importants vu l'impact des attaques malicieuses de plus en plus évoluées ainsi que les capacités de calcul réduites des téléphones intelligents. Par conséquent, plusieurs travaux se sont intéressés par la détection de malwares sur les téléphones mobiles et plus précisément les systèmes Android tout en se basant sur l'étude des signatures malicieuses. Comme nous l'avons mentionné dans le chapitre précédent, on peut classer ces approches en deux catégories : la détection de malwares par les signatures statiques et celle basée les signatures comportementales.

Les signatures statiques des applications malicieuses sont largement utilisées dans les antivirus commerciaux vu la rapidité de détection de malwares connus. Ces signatures peuvent être générées à partir du calcul de l'empreinte de hachage du code du malware. Un exemple de signatures de hachage de malwares connus sur le système Android est illustré dans le tableau

2.1. Cependant ces signatures peuvent être facilement contournées par les techniques d'obfuscation qui consistent à apporter des modifications sur le code sans affecter son fonctionnement (par exemple renommer les variables ou les fonctions, ajouter des lignes de codes sans effets, renommer les packages Android, etc.). Selon Rastogi *et al.* (2013) ce type de détection peut être aussi contourné à travers le cryptage du bytecode des applications Android ainsi que les fichiers .dex contenant le code des applications sous format Smali.

Tableau 2.1 Expemle de signatures de hachage de malwares Android
Tiré de Isohara *et al.* (2011)

Malware	**Nom du package**	**Signature SHA-1**
DroidDream	Com.droiddream.bowlingtime	72adcf43e5f945ca9f72 064b81dc0062007f0fbf
Geinimi	Com.sgg.spp	1317d996682f4ae4cce6 0d90c43fe3e674f60c22
Fakeplayer	Org.me.androidapplication1	1e993b0632d5bc6f0741 0ee31e41dd316435d997

Plusieurs travaux comme (Ping *et al.*, 2014), (Talha *et al.*, 2015) et (Qin *et al.*, 2013), se sont intéressés par un autre type de signatures statiques malicieuses qui est basé sur l'étude des permissions des applications Android. En effet, les permissions accordées à une application données peuvent donner une idée préalable sur les risques potentiels de comportements malicieux. Pour extraire ces permissions, le recours à un serveur externe est inévitable pour l'assemblage et le désassemblage des applications. Une étude de ces permissions combinée avec une étude sémantique du code de l'application permet de détecter la présence d'un éventuel malware avant son exécution. Cependant cette approche ne peut être exécutée dans le téléphone et elle est gourmande en terme de ressources de calcul.

L'étude des appels API des applications mobiles représente une autre technique de détection statique de malwares. Cette technique est présente dans divers travaux comme (Fan *et al.*, 2015), (Zou *et al.*, 2015) et (Wu *et al.*, 2012). Dans le travail de Fan *et al.* (2015), les appels API sont utilisés pour détecter les applications malicieuses dont le code est enfoui dans des applications normales. Ils étudient donc les appels API d'applications non malicieuses et forment un modèle de comportement normal traduit par les listes des appels API. Toute déviation de ce modèle est considérée comme comportement malicieux. Une illustration du modèle de détec-

tion est représentée dans la figure 2.1. Dans (Zou *et al.*, 2015), les auteurs proposent un outil de détection de malwares sur les systèmes Android qui s'appelle *DroidMat*. cet outil se base également sur l'analyse statique des applications mobiles à travers l'étude des appels API. En effet, les auteurs affirment que ces appels permettent d'informer sur les types d'opérations qui sont susceptibles d'être exécutées par une application donnée. Par exemple, certaines applications malicieuses ont tendance à extraire les informations sensibles du téléphone comme le numéro de série à travers l'appel en arrière-plan de l'API *getDeviceId()*. Pour les applications normales, cet appel ne se fait pas en arrière-plan et est affiché à l'écran du téléphone.

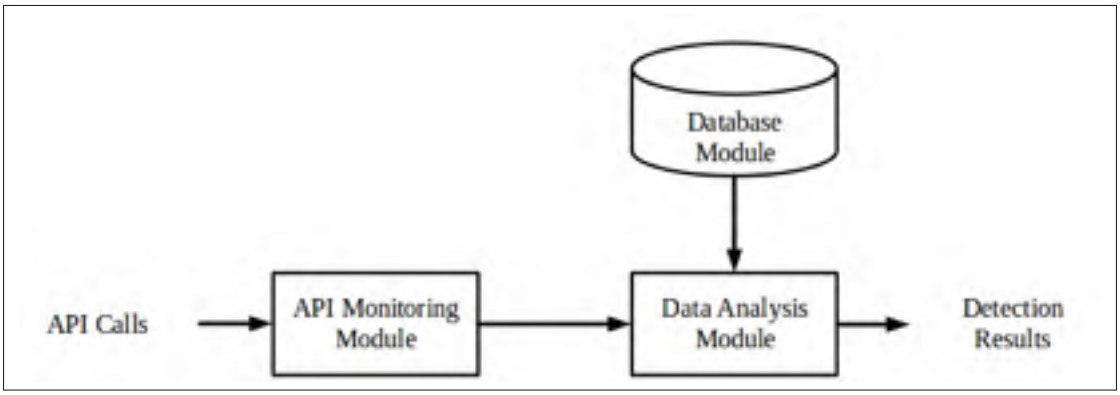


Figure 2.1 Modèle de détection de malwares avec les appels API
Tirée de Fan *et al.* (2015)

Une méthode de détection de malwares sur les systèmes Android a été proposée par Wang et Wu (2015). Dans ce travail on utilise des signatures hiérarchiques qui combinent à la fois les signatures des appels API, les signatures de hachage des applications malicieuses, ainsi que celle des classes et les méthodes de ces applications. Une description de cette approche est illustrée dans la figure 2.2. La combinaison des deux techniques de détection (permission et appels API) est aussi exploitée par plusieurs travaux comme Sharma et Dash (2014), Peiravian et Zhu (2013), Chan et Song (2014), Zeng *et al.* (2014), et Aysan et Sen (2015). Dans ces travaux on combine la robustesse des deux outils de détection pour augmenter le niveau de performance du système de détection en général. L'utilisation d'algorithmes de machine learning est présente dans ces travaux pour classifier les applications analysées.

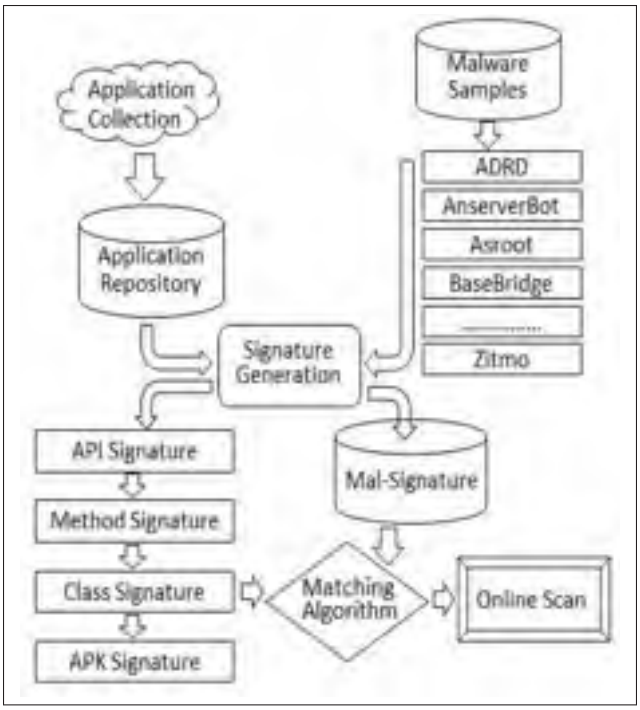


Figure 2.2 Signature hiérarchique combinée d'applications malicieuses
Tirée de Wang et Wu (2015)

Le comportement malicieux des malwares sur les téléphones mobiles peut être également traduit par l'analyse statique du code malicieux et l'extraction des graphes d'appels de fonctions. Ce graphe illustre la structure et la logique des appels de fonction d'un code malicieux et les organise sous forme de graphe. Un exemple de graphes d'appels de fonctions du malware "*Android :RuFraud-C*" appartenant à la famille malicieuse "*FakeInstaller*" est illustré dans la figure 2.3 tirée de Gascon *et al.* (2013). Les noeuds foncés indiquent les structures d'appel de fonctions malicieuses détectées par cet outil. Quant aux autres noeuds, ils représentent les structures d'appels normaux.

Le deuxième type de détection par signatures de malwares est la détection à travers les signatures comportementales. Ce type de détection se fait au moment d'exécution des applications et s'intéresse aux comportements malicieux. L'interception et l'analyse des appels systèmes entre les applications et le noyau du système Android constituent l'un des outils les plus utilisés pour la détection comportementale de malwares mobiles. Dans Isohara *et al.* (2011), une étude du

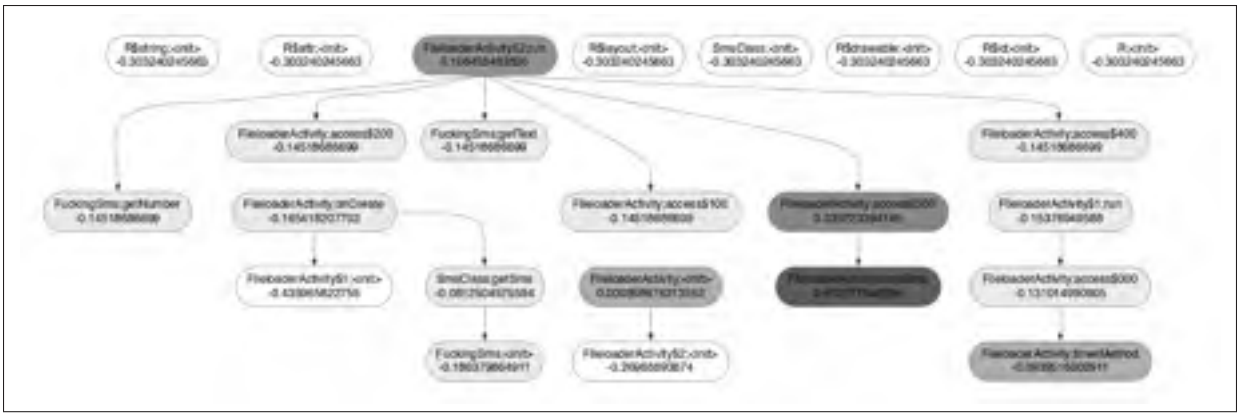


Figure 2.3 Graphe d'appels de fonctions du malware "*Android :RuFraud-C*"
Tirée de Gascon *et al.* (2013)

comportement d'applications Android à travers l'analyse des fichiers log contenant la liste des appels systèmes qui a été faite au niveau du noyau du système Android. Ces fichiers sont analysés avec les algorithmes de pattern matching pour détecter la présence de signatures d'appels systèmes déterminées au préalable. Ces signatures sont sous format d'expressions régulières dont la construction n'a pas été détaillée. Un exemple de signatures malicieuses tiré de Isohara *et al.* (2011) est illustré dans le tableau 2.2.

Tableau 2.2 Expemle de signatures d'appels système
Tiré de Isohara *et al.* (2011)

No	**Type d'attaque**	**signature**
1	Info.Leakage	^recv\(.*(%IMSI%|%SIM%).*$
2	Info.Leakage	^recv\(.*(%IMEI%|%ANDROID%).*$
3	Info.Leakage	^recv(.*(%TEL%|%MAIL%).*$
4	Jailbreak	^(execve|read)\(.*/asroot
5	Jailbreak	^(execve|read)\(.*/exploid
6	Jailbreak	^(execve|read)\(.*/rageagainstthecage
7	Jailbreak	^(execve|read)\(.*/gingerbreak
8	Abuse of root	^execve\("/(system/) ?(bin|sbin|xbin)/su"
9	Abuse of root	^open\(".*/iptables"
10	Abuse of root	^superuser(\....) ?
11	Abuse of root	^execve\(".*/busybox"
10	Abuse of root	^execve\(".*/(chmod|chown|ls\b|mkdir|rmdir)"

Dans (Lin *et al.*, 2013b), une étude détaillée pour la construction des signatures comportementales d'appels système a été menée dans le but de détecter les malwares sur les systèmes Android. Cette technique consiste à prendre deux types d'applications : les applications normales et les applications malicieuses groupées par famille de malwares. Pour chaque famille on extrait les séquences d'appels système et on ne garde que les séquences qui sont communes. Une étape de filtrage des séquences doit ensuite être faite en considérant de plus les séquences d'appels systèmes des applications normales. Une description détaillée de cette technique va être établie dans le chapitre suivant. D'après Lin *et al.* (2013b), une précision de détection allant jusqu'à 95.97% a pu être obtenue avec ce type de signatures. Cependant les performances de détection restent toujours dépendantes de la phase de construction des signatures et la capacité à trouver des applications appartenant aux mêmes familles de malwares.

D'après cette revue de littérature, nous pouvons conclure que bien que les signatures statiques de malwares permettent de détecter des malwares avant leur exécution, ces signatures souffrent tout de même de plusieurs inconvénients. Elles sont dans la plupart des cas irrésistibles contre les techniques d'obfuscation et de polymorphisme de malwares. De plus, certains types de signatures statiques nécessitent des techniques d'instrumentation du code (assemblage et désassemblage du code pour extraire l'information pertinente) dans un serveur externe. En ce qui concerne les signatures comportementales, elles permettent de détecter la présence d'un éventuel malware en cours d'exécution et sont résistibles aux techniques d'évasion de la détection mentionnées au préalable. Cependant, la phase de construction de ces signatures reste délicate et nécessite des améliorations surtout que les malwares mobiles sont en perpétuelle évolution. Cette évolution augmente la complexité de ces signatures en terme de taille et de construction. L'accélération de la détection de malwares mobiles et le compactage de ces signatures sont donc nécessaires pour rendre ce type de détection plus efficace.

2.3 Accélération du pattern matching avec le calcul parallèle

Les algorithmes de pattern matching sont largement utilisés dans les systèmes informatiques vu la multitude de domaines d'applications (systèmes de détection d'intrusions, imagerie, la

bio-informatique, etc.). Le développement des plateformes de calcul parallèle a rendu possible l'accélération de tels algorithmes. Plusieurs sont les travaux qui se focalisent sur les traitements de pattern matching parallèles et ses différents types comme Aho-corasick, Boyer et Moore et ses extensions ou variantes. Des techniques d'accélération matérielles et logicielles ont été exploitées dans plusieurs travaux. Les algorithmes de correspondance de patrons parallèles basés sur des architectures matérielles (Van Lunteren *et al.* (2006), Scarpazza *et al.* (2008), Vasiliadis *et al.* (2011)) utilisent principalement les GPUs, les circuits FPGA ainsi que les processeurs Cell/B.E de IBM. Cependant ces techniques souffrent d'un problème majeur. En effet, avec ce type d'implémentation le coût de maintenance et de mise à l'échelle devient problématique surtout avec les bases de données de référence qui changent et évoluent souvent. Les algorithmes de pattern matching parallèles basés sur les architectures logicielles utilisent quant à eux les processeurs multicœurs pour accélérer le traitement.

En effet, Tran *et al.* (2014) ont proposé une implémentation parallèle de l'algorithme Aho-Corasick optimisée pour l'architecture Kepler de Nvidia. Cette approche utilise efficacement la technologie Hyper-Q. Elle a pour but d'augmenter le nombre de connexions entre le host et le device (CPU et GPU) en établissant 32 connexions hardwares parallèles. Ainsi chaque processus utilisant le paradigme MPI peut être assigné à une queue séparée, maximisant l'utilisation de la GPU. Cette technique est illustrée dans la figure 2.4. Avec l'implémentation de la version parallèle de AC optimisée pour cette architecture de Tran *et al.* (2014), on atteint un débit d'exécution allant jusqu'à 450 Gbps sur une GPU Nvidia Tesla K20. Comparée à la version séquentielle, une accélération de 1.45x est obtenue avec cette implémentation.

Cheng-Hung Lin et Shyu (2013) se sont intéressés par la résolution de problèmes de parallélisations classiques des algorithmes de pattern matching. En effet, d'une façon générale, pour implémenter un système de correspondance de patron parallèle il faut segmenter le flux de données à analyser et allouer un thread pour chaque segment. Chaque thread doit alors parcourir le segment de données et vérifier la présence d'un des patrons de référence. Cependant avec une telle technique il est impossible de détecter les patrons au niveau de la frontière de deux segments successifs. Même l'extension du champ de recherche de chaque thread génère un coût de

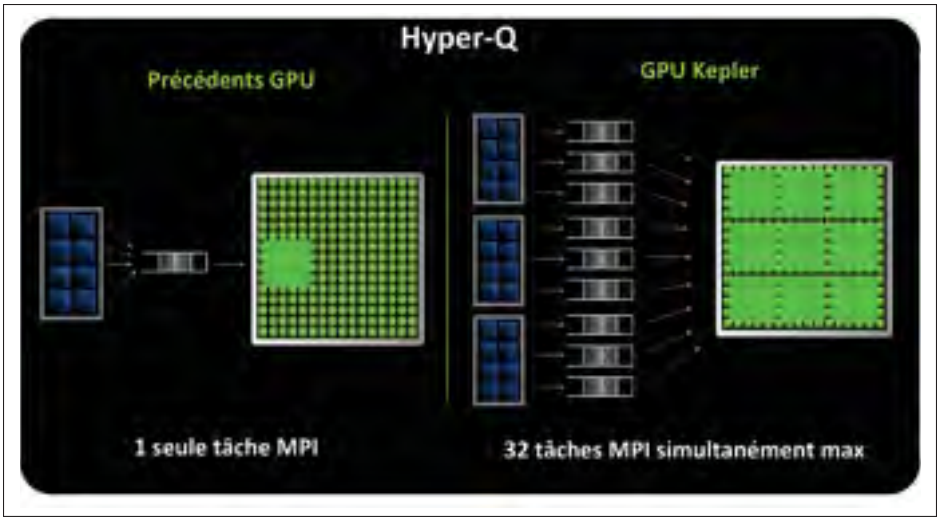


Figure 2.4 Technologie Hyper-Q dans les GPU kepler de NVIDIA

calcul supplémentaire qui va ralentir la performance du système parallèle. Pour remédier à ce problème, Cheng-Hung Lin et Shyu (2013) ont proposé un algorithme de pattern matching parallèle appelé PFAC qui repose sur l'utilisation des structures d'automates déterministes finis. L'idée clé de cet algorithme est d'allouer un thread pour chaque élément du flux de données à analyser et non pas pour chaque segment. Chaque thread commence la recherche depuis sa position initiale et termine son exécution s'il s'agit de transition d'échec. Une implémentation de cet algorithme a été élaborée sur une GPU classique. Les auteurs affirment qu'avec PFAC une accélération de 4000x a été obtenue par rapport à la version séquentielle de l'algorithme AC ainsi qu'une accélération de 3x comparé à d'autres travaux de pattern matching parallèles (comme (Huang *et al.*, 2008), (Schatz et Trapnell, 2007) et (Vasiliadis *et al.*, 2009)).

Vasiliadis et Ioannidis (2010) ont proposé un anti-malware massivement parallèle basé sur les algorithmes de correspondance de patrons appelé Gravity. C'est la version modifiée de l'anti-malware commercial ClamAV qui offre des services de détection en ligne. Pour accélérer le traitement, Vasiliadis et Ioannidis (2010) proposent d'implémenter un filtre parallèle sur une GPU qui ne va considérer que le préfixe des entrées à analyser. Si le préfixe correspond avec l'un des préfixes des signatures malicieuses, cette entrée va être envoyée vers la CPU pour qu'il continue la recherche. Dans le cas échéant, l'entrée est considérée comme non malicieuse. La

figure 2.5 illustre l'architecture générale de Gravity. Avec cet outil, une accélération de 100 fois a été obtenue comparée à ClamAV exécuté séquentiellement sur une CPU. Un débit allant jusqu'à 40 Gbit/s est également obtenu sur une GPU NVIDIA GeForce GTX295.

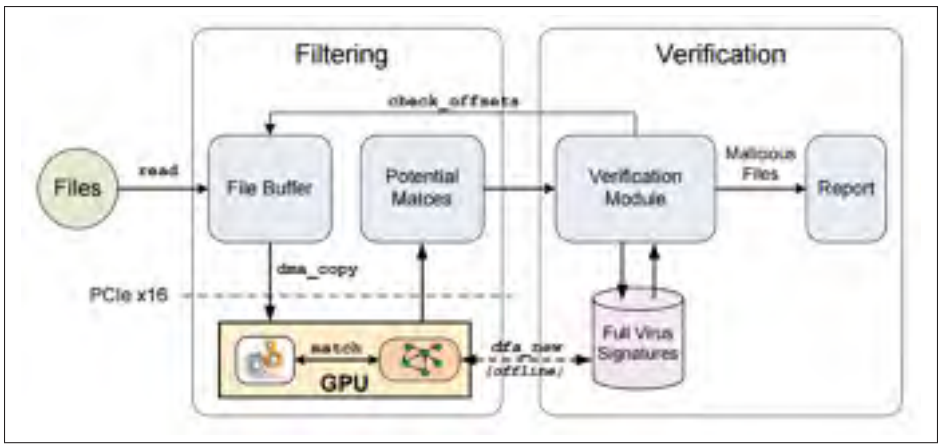


Figure 2.5 Architecture de détection parallèle de Gravity
Tirée de Vasiliadis et Ioannidis (2010)

Le travail présenté dans Zha et Sahni (2011) a porté sur la mise en œuvre de l'algorithme Aho-Corasick sur une GPU. L'arbre du Aho-Corasick a été précalculé sur la CPU et a été stocké dans la mémoire cache de la GPU. La chaîne d'entrée à analyser a été placée dans la mémoire globale de GPU puis partitionnée sur différents blocs et allouée à plusieurs threads. Zha et Sahni (2011) ont utilisé un certain nombre d'optimisations afin d'améliorer encore les performances de la mise en œuvre de l'algorithme. En effet, le groupe de recherche a exploité la conversion de type ou *casting* en anglais de la chaîne d'entrée de *unsigned char* vers *int4* pour faire en sorte que chaque thread va lire 16 caractères à la fois de la chaîne entrée de la mémoire globale au lieu d'un seul caractère. Afin d'améliorer encore l'utilisation de la bande passante, les accès en lecture des threads du même *wrap* ont été fusionnés diminuant ainsi le temps de latence de l'algorithme. Une GPU NVIDIA Tesla GT200 a été utilisée offrant une accélération de 9.5x par rapport à la version séquentielle de l'algorithme. Il est à noter que les optimisations offertes par ce travail ne sont applicables que pour la plateforme Cuda.

Yang et Prasanna (2013) proposent quant à eux une approche parallèle de détection d'intrusion dans le réseau. Cette approche est basée sur l'algorithme de correspondance de patron Aho-Corasick qu'ils appellent *head-body finite automaton*. La technique proposée possède la particularité de diviser l'automate en deux parties : une première partie qui contient les préfixes des patrons de l'automate et une seconde partie qui contient le reste. Cette approche a été implémentée sur une CPU multicoeurs et a pour but d'augmenter le débit d'exécution du système à travers l'augmentation du ratio de correspondance.

Une autre approche de parallélisation d'algorithme de pattern matching est proposé par Arudchutha *et al.* (2013). L'idée clé de leur approche est de prendre une version parallèle de l'algorithme AC sans considérer les transitions d'échec et de diviser la base de données de patrons de référence en plusieurs sous-ensembles. Pour chaque sous-ensemble, on construit l'automate déterministe fini correspondant. On alloue à chaque automate un thread qui va s'occuper de la recherche de patrons dans l'automate qui lui est associé. La majeure motivation de cette approche réside dans le fait que le temps de construction de l'automate dans la phase de prétraitement est considérable, surtout que le nombre de patrons mis en jeux est énorme. L'architecture proposée est illustrée dans la figure 2.6. Cette approche a été implémentée sur un CPU AMD Opteron à 8 cœurs et a donné des résultats meilleurs que dans l'approche de Herath *et al.* (2012) où la même technique est utilisée, mais avec une base de données de patrons plus petite.

Dans Pungila et Negru (2012), les algorithmes Aho-Corasick et Commentz-Walter ont été utilisés pour effectuer l'analyse accélérée d'antivirus par une GPU. L'implémentation de ces algorithmes était représentée dans la GPU utilisant les piles. Cette approche se distingue par deux techniques : une technique de sérialisation de l'automate dans une mémoire continue ainsi qu'une technique de substitution de la fonction qui donne l'état suivant par des mécanismes de décalage. La parallélisation de ces algorithmes a été réalisée en appliquant une approche de parallélisme de données.

Une implémentation parallèle de l'algorithme de correspondance de patrons Knuth-Morris-Pratt Mandumula (2011) a été proposée dans Bellekens *et al.* (2013). Cette implémentation

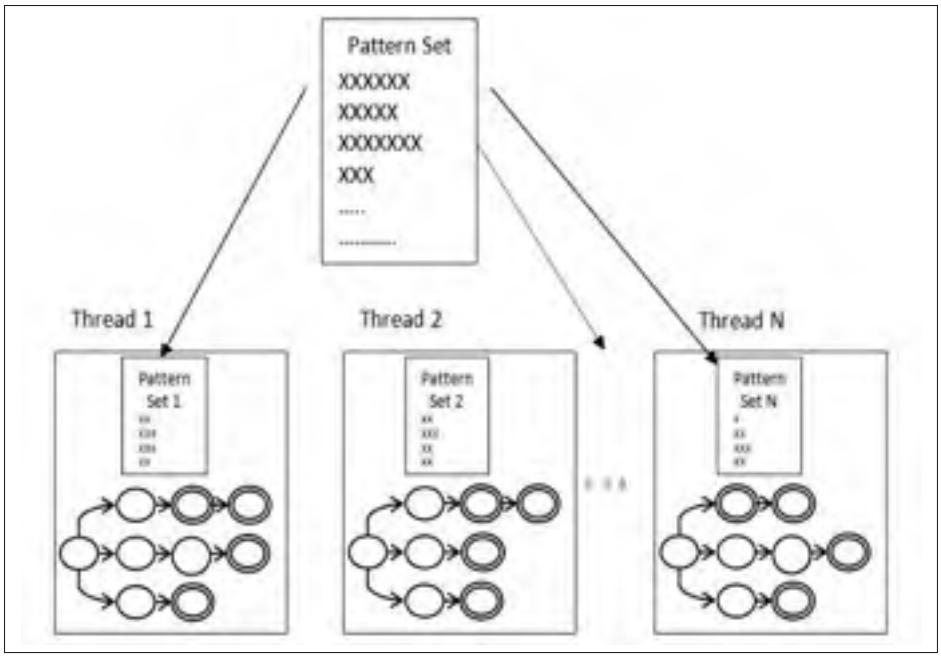


Figure 2.6 Subdivision de la dataset et allocation des threads
Tirée de Arudchutha *et al.* (2013)

parallèle a été exploitée pour la détection d'intrusions dans le réseau en utilisant les performances de calcul parallèle des GPUs. Une première technique d'accélération de traitement parallèle utilisée est le *loop splitting*. Cette technique consiste à segmenter une partie itérative du programme contenant plusieurs instructions indépendantes. Ceci à pour but de diminuer la charge sur les registres des kernels de la GPU et assurer un niveau plus élevé de convergence de threads. Une autre technique d'accélération de calcul parallèle exploitée dans ce travail de recherche est l'utilisation de la mémoire globale de la GPU pour stocker les patrons ainsi que la table de décalage. Cette table contient le nombre de décalages à effectuer en cas de transition d'échec. Une accélération d'environ 30x est obtenue avec cette implémentation cependant elle reste moins performante que celle de l'algorithme PFAC.

Une variante parallèle de l'algorithme Wu-Manber appelée *Simplified Wu-Manber* (SWM), a été mise en oeuvre par Vespa et Weng (2012) en utilisant l'API OpenCL pour la détection d'intrusion dans le réseau. L'algorithme a été modifié à travers la diminution de la taille des blocs de patrons ainsi que l'élimination du traitement de hachage. Avec ces changements, la

taille des tables de décalage a été réduite. Par conséquent, il était possible de les placer dans la mémoire partagée de la GPU. En outre, le nombre de comparaisons requis pour le traitement de la chaine de données en entrée a été réduit et par conséquent le niveau de la divergence de threads dans les kernels de la GPU a été évité. Enfin, les calculs de hachage ont été entièrement éliminés, ce qui améliore encore la performance de l'algorithme. Les tables de décalage de la variante de Wu-Manber ont été calculées par la CPU et ont été transférées ensuite à la mémoire partagée de la GPU. La chaîne d'entrée sous forme de paquets réseau a été enregistrée au niveau de la mémoire de l'hôte appelée *pinned memory*. Ce type de mémoires permettent au GPU d'accéder directement aux données sans transiter par la CPU. L'allocation et la désallocation de cette mémoire sont certes couteuses cependant l'accès à ce type de mémoire est très rapide et son utilisation devient avantageuse dans le cas de transferts fréquents de données de taille importante.

D'après notre revue de littérature, il n'y a aucun travail de recherche qui se focalise sur l'accélération des algorithmes de pattern matching sur les GPUs mobiles ou même sur l'accélération des traitements relatifs à la détection de malware sur les téléphones mobiles en général. Ceci nous ramène au besoin de bien choisir et de filtrer les techniques d'optimisation de ces algorithmes qui peuvent être supportées par les GPUs mobiles et de voir comment on peut adapter ces algorithmes pour accélérer la détection des malwares mobiles.

2.4 Techniques de compactage de données

La plupart des algorithmes de correspondance de patrons utilisent une structure d'automate générée à partir d'une base de données de patrons références. La structure de l'automate requiert un espace de stockage important ce qui représente une contrainte majeure pour l'utilisation de tels algorithmes dans les systèmes embarqués possédant une capacité de stockage limitée.

Dans la littérature, il existe divers techniques de compactage de données relatives aux algorithmes de correspondance de patrons parallèles ou séquentiels. Parmi ces techniques on peut citer la compression par bitmap qui est largement adoptée dans plusieurs travaux. Cette tech-

nique consiste à utiliser un tableau de bits de longueur N égale à la taille de l'alphabet utilisé (soit 256 si on utilise l'alphabet du code ASSCI) pour indiquer la présence d'états valides (un bit 1 à une position i indique la présence d'un état valide en ayant comme entrée l'alphabet i et un bit 0 indique la présence d'état d'échec). L'un des problèmes majeurs de telle technique est le fait qu'au pire des cas on doit associer deux références mémoires pour chaque caractère ainsi que 256 bits pour chaque bitmap ce qui ne réduit pas considérablement l'espace total requis pour le stockage de la structure de l'automate.

Une autre de technique de compression associée aux algorithmes de correspondance de patrons utilisant la structure d'automate est la compression de chemins. Cette technique a été exploitée dans plusieurs travaux dans la littérature et qui s'intéressent particulièrement à la détection d'intrusions. Comme c'est illustré dans la figure 2.7, cette technique consiste à regrouper les états consécutifs de l'automate ayant une seule transition possible en un seul état compressé. Les transitions dans cet état sont stockées dans une liste linéaire chainée ainsi que les transitions d'échec pour chaque caractère. Cependant l'un des problèmes majeurs d'une telle technique c'est la gestion des transitions d'échec. En effet, ce type de transitions qui est relatif à un état donné peut pointer sur un autre état qui est déjà compressé. La gestion des transitions d'échec devient alors plus complexe à gérer.

Pungila (2013) ont combiné ces deux dernières techniques et proposé un algorithme hybride optimisé pour le traitement parallèle au sein d'un système de détection d'intrusion. En effet, l'automate est représenté par une liste de nœuds qui intègrent à la fois la structure des bitmaps ainsi que les chemins compressés des nœuds. En outre, pour pallier au problème de gaspillage de la pile mémoire (miettes d'espaces mémoire non utilisées) lors de l'allocation des nœuds, tous les nœuds fils relatifs à un nœud courant sont stockés dans des zones mémoires consécutives. Pungila (2013) ont utilisé dans leurs expérimentations les signatures de malwares de ClamAV et ont comparé le taux de compression des trois techniques. En effet, d'après les résultats expérimentaux, l'application de la technique de compression de chemins à l'automate offre une réduction de la taille de 65% par rapport à la version non optimisée. Ensuite, l'application de la technique du bitmap seule à l'automate offre un taux de réduction en mémoire moins in-

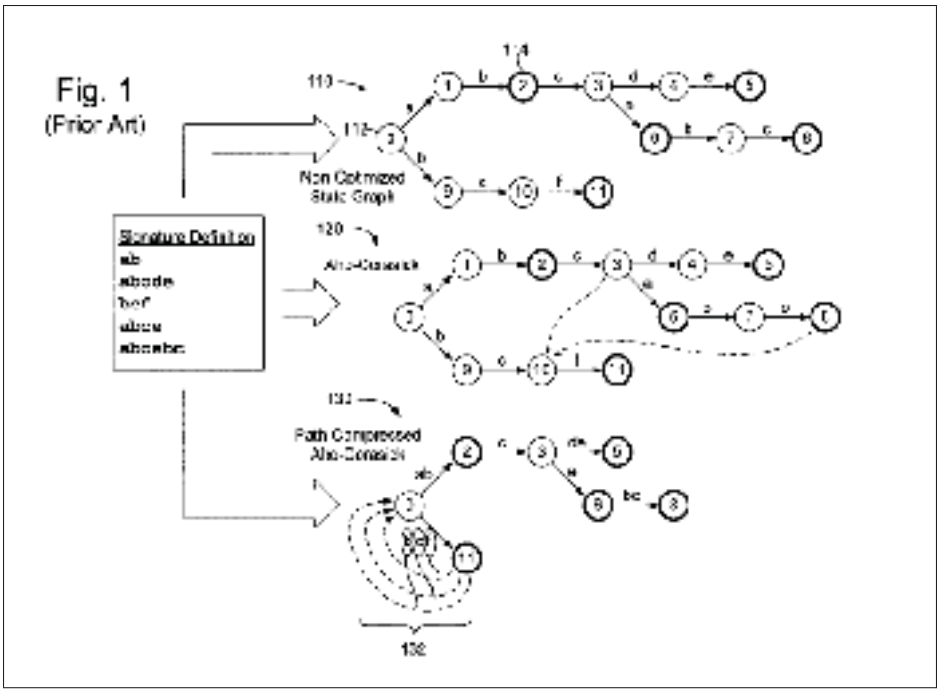


Figure 2.7 Compression des chemins de l'automate
Tirée de Venkatachary (2009)

téressant et égal à 27% par rapport à la version originale non optimisée. Enfin, ils affirment que leur approche proposée qui consiste à combiner les deux techniques offre un résultat meilleur : un taux de réduction de l'utilisation de la mémoire égale à 78% par rapport à la version non optimisée.

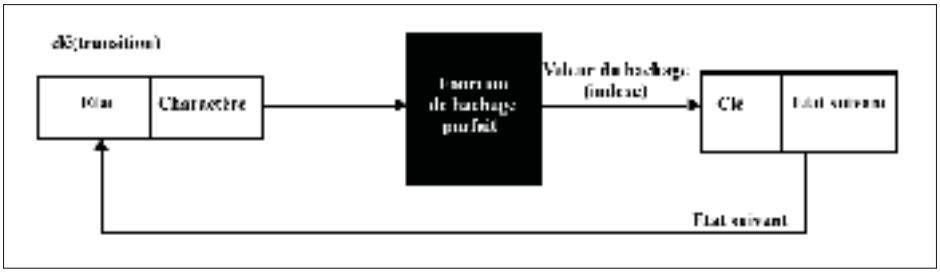


Figure 2.8 Le hachage parfait
Tirée de Lin *et al.* (2012)

Vespa *et al.* (2009) et Vespa et Weng (2011) ont introduit un algorithme de réduction d'automates des algorithmes de pattern matching appelé P3FSM. C'est l'acronyme de *Portable pre-*

dictive pattern matching finite state machine. Ils ont introduit cet algorithme pour la réduction de l'espace de stockage requis pour les signatures des virus relatifs aux systèmes de détection d'intrusion. Une description détaillée de cet algorithme va être élaborée dans le chapitre suivant. Les résultats expérimentaux obtenus par les chercheurs affirment que cet algorithme offre un taux de compactage intéressant (80x) ainsi qu'un temps d'exécution réduit. Ils affirment également que cet algorithme peut être utilisé efficacement au niveau des systèmes de détection d'intrusions parallèles ayant des contraintes de mémoire limitée.

Lin *et al.* (2012) proposent une autre technique de compactage d'automates appelé hachage parfait. L'idée de base de cet algorithme consiste à prendre la matrice de transitions de l'automate déterministe à états finis et utiliser une fonction de hachage pour ne stocker que les transitions valides dans une table de hachage. La figure 2.8 illustre le principe de base de cette approche. Une fonction de hachage parfait donne à partir de chaque clé (transition valide) une valeur de hachage unique et sans collisions. Cependant, le calcul de ces valeurs peut dégrader la performance du système. C'est pour cette raison qu'il faut bien choisir une fonction de hachage efficace, qui ne requière pas un calcul gourmand. L'élaboration de cet algorithme passe par trois étapes : la construction de la matrice des clés, la construction des tables de hachage et de décalages et enfin le déclenchement du processus d'analyse. Pour la phase d'expérimentation Lin *et al.* (2012) utilisent les paquets DEFCON qui contiennent une quantité importante de patrons d'attaques réseau réelles. Ils utilisent la technique de compactage pour différentes GPUs et affirment que le taux de compactage avec le hachage parfait obtenu par rapport à la version non optimisée est égale à 99%. Ce taux de compression est certes intéressant cependant l'algorithme requière un temps de calcul important pour élaborer le processus de compactage.

2.5 La cryptographie parallèle

La plupart des algorithmes de cryptographie requièrent des calculs arithmétiques utilisant des structures mathématiques différentes. Dans les systèmes de cryptographie modernes, nous avons besoin en général d'exécuter des calculs dans des architectures finies, ce qui rend indispensable l'utilisation d'une arithmétique modulaire efficace. Cependant, ces opérations sont

très gourmandes en terme de ressources de calcul vu leur complexité et la taille des données mises en jeu.

En effet, la taille des clés exploitées dans les algorithmes de cryptographie rend les fonctions arithmétiques plus coûteuses en termes de temps d'exécution. Par ailleurs, on trouve aujourd'hui plusieurs architectures grand publiques intégrant plusieurs unités de calcul qui sont réparties sur les processeurs ainsi que sur les cartes graphiques. La popularité de ces ressources et leur facilité d'exploitation ont attiré plusieurs recherches pour accélérer les algorithmes de cryptographie.

Par exemple, une étude comparative concernant l'implémentation parallèle de l'algorithme RSA sur une CPU et une GPU a été élaborée par Zhang *et al.* (2012). Les auteurs affirment que l'implémentation parallèle avec une GPU a donné de meilleurs résultats comparés à l'implémentation sur la CPU. Un taux d'accélération égale à 45 fois par rapport à la version parallèle sur la CPU a été atteint. D'après cette recherche, les auteurs affirment que les GPUs sont mieux adaptées pour les processus parallèles simples, ils ont une faible consommation d'énergie par rapport aux CPUs et peuvent être utilisés efficacement pour les algorithmes de cryptographie.

Une autre implémentation parallèle de RSA a été élaborée par Yang *et al.* (2015). Dans ce travail, on propose une approche de parallélisation de la multiplication modulaire appelée multiplication de Montgomery qui permet d'accélérer le traitement de l'algorithme de cryptographie RSA. Une autre technique d'accélération a été également utilisée dans ce travail qui consiste à utiliser la fonction *wrap shuffle* des GPUs Nvidia qui permettent à un ensemble de threads de communiquer entre eux sans faire recours à la mémoire partagée. Ceci réduit considérablement le temps de latence de l'algorithme et permet d'améliorer sa performance.

L'importance de l'algorithme de cryptographie symétrique AES et le vaste domaine d'application de cet algorithme a attiré plusieurs travaux de recherche pour l'accélérer à travers le calcul parallèle. En effet, Nagendra et Sekhar (2014) ont implémenté une version parallèle de AES sur un processeur Intel Core 2 Duo en utilisant OpenMP dans le but de réduire le temps d'exécution. Une stratégie de parallélisation qui repose sur le principe de diviser pour régner

a été adoptée. Une amélioration de 40% à 45% dans la performance du AES parallèle a été atteinte comparée à la version séquentielle.

Dongara et Vijaykumar (2003) ont décrit une implémentation parallèle de l'algorithme AES. Ce dernier a été divisé en deux parties : une partie parallélisable et une autre non parallélisable. Les auteurs ont montré que les boucles itératives incluses dans la plupart des fonctions de l'algorithme AES consomment plus de temps (ces boucles sont responsables du cryptage des blocs de données et le décryptage). Ils montrent également que certaines boucles sont entièrement parallélisables. Dans ce but, ils ont fait quelques transformations au niveau des boucles itératives pour accélérer le traitement. Cependant, l'accélération de leur implémentation dépend des méthodes utilisées pour la lecture et l'écriture de données à traiter ainsi que des boucles itératives non parallélisables et qui prennent un temps d'exécution considérable. Un gain de performance de 3.5x est obtenu avec ce type de traitement par rapport à la version séquentielle du AES.

La problématique d'accélération stockage des données sécurisées dans les téléphones mobiles avec le mécanisme de cryptographie fut la motivation majeure du travail de recherche de Alomari et Samsudin (2011a). En effet, dans cette proposition une variante de l'algorithme AES appelée XTS-AES a été implémentée en parallèle sur une GPU mobile avec OpenGL. Une implémentation parallèle de l'algorithme Blowfish a été également proposée par Davis *et al.* (2015) en utilisant la plateforme RenderScript des téléphones Android. Cette implémentation est portable et sa performance a été testée sur différents téléphones intelligents comme Nexus 5, Nexus 10 et Samsung Galaxy S4. Une accélération de 3x a été enregistrée par rapport à la version séquentielle cependant cette implémentation souffre d'un temps de latence important.

Une étude de la performance des algorithmes de cryptographie à base de courbes elliptiques (ECC) sur les plateformes mobiles a été menée par Vanderlei de Arruda *et al.* (2015). Les auteurs affirment que ce type d'algorithmes est considéré parmi les protocoles les plus gourmands en termes de temps de calcul. Les ECC peuvent être également utilisées pour chiffrer les accès aux applications en ligne depuis les téléphones mobiles. Pour optimiser le calcul cryp-

tographique, plusieurs types d'algorithmes de multiplication modulaires parallèles ont été implémentés et comparés comme l'algorithme de Montgomery parallèle (Baktir et Savaş, 2013), Bitpartite (Kaihara et Takag, 2008) et le Multipartite (Giorgi *et al.*, 2013).

Des solutions matérielles ont été également proposées pour accélérer les protocoles de cryptographie. Par exemple Guo *et al.* (2012) ont conçu un processeur spécialisé pour le traitement parallèle de l'algorithme RSA. L'architecture de ce processeur accélère le traitement de l'algorithme en supportant de plus l'exécution optimale et parallèle de l'algorithme de multiplication modulaire de Montgomery. Un coprocesseur spécialisé pour l'exécution parallèle de l'algorithme de cryptographie à base de courbes elliptiques a été proposé par MuthuKumar et Jeevananthan (2010). Les auteurs affirment que ce coprocesseur offre un débit d'exécution élevé et une consommation d'énergie réduite. Certains chercheurs ont également essayé de tirer bénéfice des FPGA qui offrent des unités de traitement spécialisées et entièrement programmables. Parmi ces travaux on peut citer Bora et Czajka (1999) et MuthuKumar et Jeevananthan (2010) qui ont proposé des structures de FPGA spécialisés pour l'exécution parallèle de l'algorithme RSA. Les solutions logicielles pour accélérer les protocoles de cryptographie sont certes intéressantes. Cependant leur intégration aux architectures matérielles traditionnelles reste encore problématique.

L'accélération des algorithmes de cryptographie avec le calcul parallèle fut donc l'objet de plusieurs travaux vu la complexité de ces algorithmes et leur utilisation courante que ce soit sur les téléphones mobiles ou sur les ordinateurs de bureau. D'après cette revue, les techniques d'accélération qui se trouvent dans la littérature peuvent être classées en deux catégories : accélération matérielle et accélération logicielle. La comparaison entre les différents protocoles de cryptographie parallèle devient de plus en plus difficile vu la diversité des plateformes utilisées et la différence de capacités de calcul des unités de calcul parallèles exploitées dans les travaux de recherche mentionnés.

2.6 Les Clusters

Plusieurs sont les travaux qui se focalisent sur l'utilisation du cluster pour la sécurité en général. Cependant peu de travaux dans la littérature utilisent ces architectures hautement parallèles pour l'accélération du pattern matching.

Parmi ces travaux on peut citer Tumeo et Villa (2010) qui ont introduit une version parallèle de l'algorithme AC sur un cluster de GPUs pour l'utilisation des applications d'analyse d'ADN. Ils ont développé une application maître/esclave, où le processus maître MPI est responsable de la répartition des tâches aux différents nœuds et GPUs. Le calcul parallèle au niveau des GPUs est basé sur la segmentation des données à analyser en plusieurs blocs et en attribuant à chaque bloc un seul thread du kernel CUDA. Pour détecter les patrons qui se trouvent au niveau de la frontière des blocs,les morceaux peuvent se chevaucher sur une longueur égale au patron le plus long du dictionnaire utilisé.

Tumeo *et al.* (2012) ont étendu leur travail précédent et implémenté le même algorithme sur des architectures de clusters hétérogènes et hautement parallèles. Ils ont utilisé à cette fin un super ordinateur Cray XMT, des processeurs Xeon 5560, des multiprocesseurs x86 ainsi que de GPUs NVIDIA Tesla. Une analyse de la performance des différentes implémentations de l'algorithme sur ces systèmes est également élaborée. Un maximum de performance est fourni par la machine Cray XMT avec un maximum de 28Gbps (utilisation de 128 processeurs).

Une étude comparative entre plusieurs algorithmes parallèles de pattern matching a été faite par Kouzinopoulos *et al.* (2012). Des versions parallèles des algorithmes Commentz-Walter, Wu-Manber ainsi que Set Backward Oracle Matching ont été implémentées sur une architecture de cluster constituée de 10 ordinateurs interconnectés et utilisant le paradigme MPI. Les résultats expérimentaux ont mené à conclure que la version parallèle de l'algorithme Wu-Manber est la meilleure en terme de temps d'exécution et que la performance des algorithmes varient fortement avec le type des entrées à analyser.

Une implémentation parallèle de l'algorithme KMP a été faite par Lin *et al.* (2013a) en utilisant des GPUs formées en cluster. Dans cette implémentation, la plateforme Cuda a été choisie pour le traitement parallèle ainsi que OpenMP pour la distribution et la synchronisation des tâches. Une accélération de 97x par rapport à la version séquentielle a été obtenue par rapport à la version séquentielle.

D'une façon générale, parmi les travaux qui se sont intéressés par les clusters des processeurs multicœurs on peut citer Sharma et Kanungo (2014). Ils affirment que l'une des majeures motivations pour cibler ce type d'architectures est le rapport coût-efficacité que les clusters de processeurs multicœurs fournissent. Cependant l'un des principaux défis de telles architectures est l'acheminement ou le mappage des tâches parallèles sur les différents cœurs des microprocesseurs pour que ces ressources soient utilisées efficacement. Sharma et Kanungo (2014) ont alors évalué la performance d'un algorithme d'équilibrage dynamique de la charge allouée à chaque nœud d'un cluster de processeurs multicœurs. Ils ont proposé pour cette fin une politique de distribution de la charge sur les différents nœuds d'un cluster pour tout calcul parallèle générique. Dans l'approche proposée, les tâches parallèles sont réparties sur les différents nœuds en fonction de leurs puissances de calculs. D'après les résultats expérimentaux, cette approche permet d'avoir entre 15 et 20% de réduction du temps d'exécution total.

Le coût réduit des clusters formés à partir des systèmes embarqués ainsi que la performance de plus en plus évoluée et offerte par ce type de systèmes l'ont rendu une source attrayante pour les calculs parallèles traditionnellement implémentés dans des serveurs puissants et distribués. En effet, plusieurs travaux comme Lin et Chow (2013), Kaewkasi et Srisuruk (2014a), Loghin *et al.* (2015) et bien d'autres se sont intéressés par l'élaboration de clusters pour les Big Data formés à partir de cartes embarquées à base de processeurs ARM. Dans ces travaux, les chercheurs ont essayé de porter et étudier la performance de frameworks destinés à faciliter la création d'applications distribuées ainsi que la gestion des Big Data comme Hadoop et MapReduce. En particulier, dans Loghin *et al.* (2015) une comparaison dans ce sens a été faite entre un cluster de cartes Odroid XU et un serveur Intel Xeon. Les auteurs affirment que le cluster utilisé possède des performances semblables au serveur Intel Xeon dans certains cas d'utilisations et

qu'il 4 fois plus rentable en ce qui concerne le prix. Dans Kaewkasi et Srisuruk (2014a), une optimisation du framework Hadoop a été élaborée pour les clusters à base de processeurs ARM. Une attention particulière à l'optimisation de la performance et la consommation d'énergie a été accordée dans ce travail de recherche en utilisant quelques politiques d'optimisation spécifiques aux processeurs ARM ainsi que l'implémentation de quelques modules du framework Hadoop en langage C. Une optimisation de 2x au niveau du temps d'exécution a été obtenue comparé à leur implémentation antérieure dans Kaewkasi et Srisuruk (2014b).

Dans le travail de recherche de Ou *et al.* (2012), une comparaison de la performance d'une architecture de cluster formé à partir de processeurs ARM avec celle d'un workstation Intel X86 a été élaborée. Cette comparaison a mis l'accent sur deux axes principaux à savoir le coût et l'efficacité énergétique des deux plateformes de calcul parallèle ciblées. Le cluster mis en jeu est formé à partir de quatre cartes Pandaboard sur lesquelles plusieurs types de traitements sont distribués. Ou *et al.* (2012) affirment que l'utilisation du cluster de processeurs ARM est avantageuse pour les traitements parallèles légers ainsi que les applications de transcodage des vidéos, les serveurs web et les traitements parallèles raltifs aux bases de données. Une efficacité énergétique entre 1.21 et 9.5 est obtenue grâce à l'utilisation des clusters des cartes Pandaboards par rapport au workstation d'Intel.

D'après cette revue de littérature, nous pouvons conclure que les architectures de clusters sont largement utilisées pour accélérer les traitements qui sont généralement gourmands en termes de mémoire et de temps de calcul. Une attention particulière vers les clusters formés de systèmes embarqués à ressources limitées est devenue de plus en plus importante vu les performances de calcul émergentes qu'ils offrent ainsi que le coût réduit de tels systèmes. Le mappage des tâches entre les différents noeuds du cluster ainsi que l'acheminement des processus vers les coeurs des unités de traitement des noeuds restent l'un des principaux problèmes qui déterminent la puissance de calcul de tels systèmes. L'efficacité énergétique des clusters formés de systèmes à ressources limitées reste encore problématique et à améliorer.

2.7 Conclusion

Dans ce chapitre nous avons élaboré une revue de littérature concernant les principaux axes de notre travail de recherche pour mieux cibler nos contributions et apporter nos choix.

D'après la revue de littérature des techniques d'accélération du pattern matching et la détection de malwares, les travaux dans la littérature sont centrés vers les systèmes traditionnels et non pas ceux à ressources limitées. L'accélération de ce type de traitements dans les GPUs mobiles par exemple n'était pas abordée ce qui nous ramène à chercher comment on peut tirer bénéfice de ces dispositifs pour optimiser ce type de traitements.

Dans le chapitre suivant, nous allons introduire notre solution proposée pour l'accélération des traitements de la sécurité mobile par le calcul parallèle sur des systèmes à ressources limitées.

CHAPITRE 3

UTILISATION D'ARCHITECTURES PARALLÈLES POUR L'ACCÉLÉRATION DES TRAITEMENTS DE LA SÉCURITÉ MOBILE

3.1 Introduction

L'accélération des traitements de la sécurité mobile est devenue une préoccupation de plus en plus importante. Ceci est dû à deux facteurs principaux : le développement de la capacité de calcul parallèle dans ce type de systèmes d'une part, et la croissance exponentielle des attaques ciblant ces plateformes, d'autre part. Il est donc indispensable de protéger les informations sensibles au sein des téléphones mobiles à travers l'implantation de systèmes de détection de malwares ainsi que le cryptage des données dans le but de maintenir un plus haut niveau de sécurité. Comme nous l'avons détaillé au préalable, ces types de traitements sont complexes et doivent être accélérés.

Dans ce chapitre, nous allons décrire l'architecture générale de notre solution proposée pour accélérer les traitements de la sécurité mobile avec le calcul parallèle tout en prenant en considération les différentes questions de recherches précédemment discutées. Dans la construction de notre architecture, plusieurs défis se sont imposés. Le premier consiste à choisir une technique de détection de malwares mobiles efficace et cibler un algorithme de cryptographie à accélérer. Le deuxième défi consiste à bien choisir les techniques d'accélération de calcul parallèle sur les GPUs mobiles surtout que nous sommes dans un milieu à ressources limitées. Le troisième défi consiste à implanter une technique de compression de données offrant un taux de compactage intéressant pour manipuler plus de signatures malicieuses. Enfin, le quatrième défi cible la migration vers l'architecture de cluster pour augmenter la performance générale de notre architecture.

3.2 Architecture générale

Dans cette section, nous allons décrire l'architecture générale de notre plateforme représentée dans la figure 3.1. L'architecture se compose principalement de trois modules : le module de détection, le module de cryptographie et enfin le cluster.

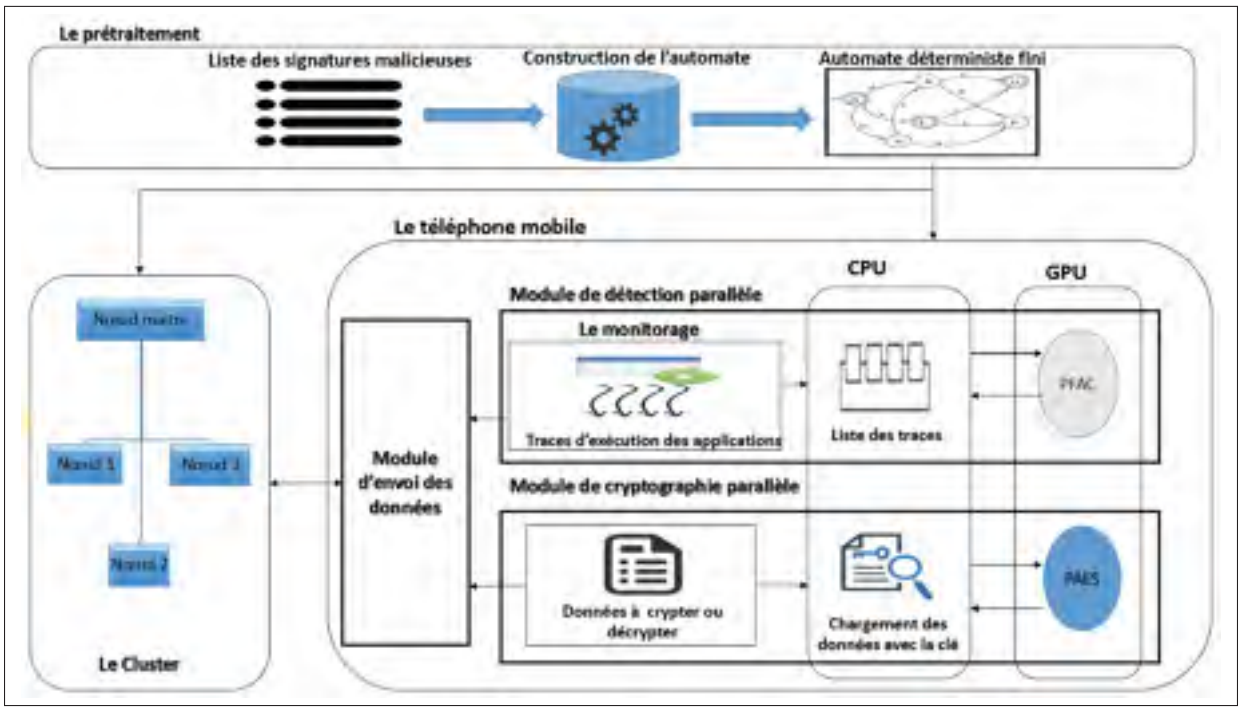


Figure 3.1 Architecture générale

3.2.1 Le module de détection

Les avantages de la détection comportementale des malwares sur les systèmes Android, discutés dans le chapitre précédent, nous a menés à choisir ce type de détection. Ce module est constitué principalement de deux parties : la partie du prétraitement située dans un hôte extérieur et la partie de traitement qui est sur le téléphone mobile.

3.2.1.1 Partie du prétraitement

Ce module est responsable de la préparation de la base de données de référence sur laquelle on va se baser et elle est située à l'extérieur du téléphone mobile. Cette base de données peut être construite à partir de patrons d'appels systèmes reflétant des comportements malicieux, des signatures de hachage de malwares, des appels API, etc. Dans notre cas nous utilisons des patrons d'appels systèmes extraits à partir du profil d'exécution de malwares comme proposé dans le travail de Lin *et al.* (2013b). Le processus de construction de la base de données des signatures sera détaillé par la suite.

La partie de prétraitement prend en entrée ces patrons et les convertit en une structure d'automate déterministe fini. Cet automate va être utilisé par la suite par le bloc de traitement parallèle pour détecter la présence d'un éventuel malware dans le système. L'avantage de l'utilisation des structures d'automates réside dans le fait qu'ils permettent de détecter la présence d'un patron malicieux dans le flux de données à analyser, et ce en une seule passe.

En général, la structure d'automate peut être représentée principalement par une matrice de transitions, une table de transitions d'échecs et les états de finaux. La figure 3.2 montre un exemple de la matrice de transitions STM (*State Transition Matrix*) que nous utilisons dans la partie de prétraitement. L'indice des colonnes reflète l'appel système en question (open, close, read, etc.), quant aux lignes ils indiquent l'état courant dans l'automate. Ainsi pour un état courant i et un appel système en entrée j la valeur de STM[i][j] indique l'état suivant vers lequel on doit être redirigé.

3.2.1.2 Partie du traitement

Le module de détection sur l'appareil mobile est constitué principalement de deux blocs : le bloc du monitorage et le bloc de traitement parallèle.

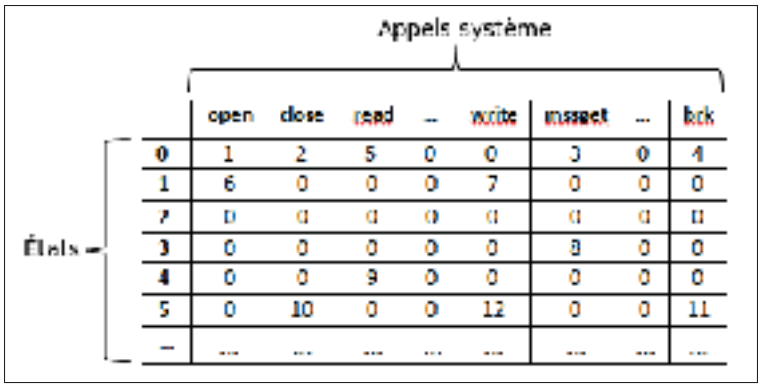


Figure 3.2 Exemple de matrice de transitions STM

A. Le monitorage

Ce bloc est responsable de la collecte des profils d'exécution des applications installées dans le téléphone. Il permet d'enregistrer les évènements et les comportements des applications en cours d'exécution à l'aide de l'extraction des différents appels systèmes émis. On utilise pour cette fin l'outil Strace. Il représente un outil de diagnostic et de débogage qui enregistre et intercepte les appels avec leurs arguments appelés par un processus donné. Seuls les appels systèmes bruts sont conservés dans les fichiers des traces d'exécution. Ces traces seront analysées en parallèle par le bloc de traitement afin de détecter un comportement malveillant.

B. Le traitement parallèle

C'est la partie centrale de notre architecture. Au niveau de ce bloc, l'analyse parallèle des traces d'exécution des applications est effectuée par la GPU. En effet, la CPU envoie les données à scanner vers la GPU mobile. Ensuite ces données seront subdivisées en des segments qui seront analysés en parallèle par les threads de la GPU qui sont organisés en blocs. L'analyse des traces se fait à l'aide d'un algorithme de correspondance de patrons parallèle qui prend en entrée les signatures malicieuses (sous format d'automate) et vérifie la présence de patrons malicieux dans la trace. Les résultats de l'analyse seront par la suite envoyés vers la CPU pour les stocker. La plateforme de calcul parallèle que nous utilisons est OpenCL. Cette plateforme

est largement utilisée pour le calcul parallèle sur le GPUs mobiles et offre un outil puissant pour accélérer les traitements sur ce type de systèmes.

L'algorithme de correspondance de patrons parallèle sur lequel se base notre framework est PFAC proposé par Cheng-Hung Lin et Shyu (2013). C'est l'acronyme de *Parallel Failureless Aho-Corasick*. Comme nous l'avons mentionné dans le chapitre précédent, PFAC représente une version parallèle de l'algorithme Aho-Corasick sans prendre en considération les transitions d'échec de l'automate. Le but de tel algorithme est de détecter en parallèle la présence de tout patron dans un flux de données à partir d'une base de données de patrons de références structurés dans une architecture d'automate. Le choix de cet algorithme a été motivé par le taux d'accélération de traitement qu'il offre et la fluidité de son implémentation. Il représente de plus une version parallèle de l'algorithme Aho-corasick dont la complexité est linéaire et qui est considéré comme référence dans le domaine du pattern matching.

Le principe du traitement parallèle pour la détection de malwares de notre framework est le suivant. Nous considérons un flux de données à analyser qui consiste en une séquence d'appels système. Les threads dans la GPU sont organisées en groupes. Chaque groupe prend un segment du flux de données. Comme indiqué dans la figure 3.3, on alloue pour chaque thread un byte du flux en entrée. Chaque thread commence le traitement depuis sa position de départ. En cas de transition valide, le thread poursuit son traitement. Dans le cas échéant, il reprend la recherche depuis sa dernière position de départ incrémentée par le nombre total des threads du bloc.

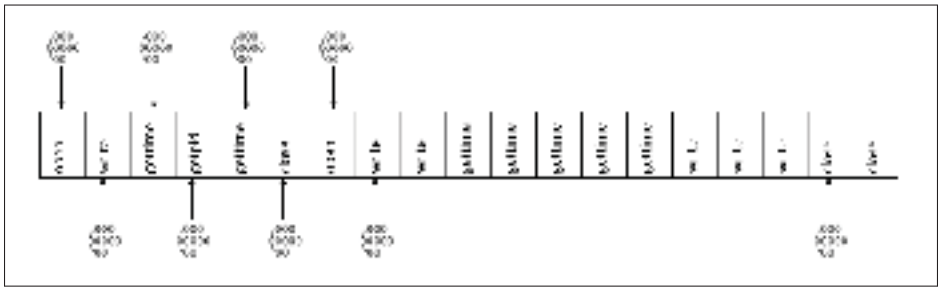


Figure 3.3 Allocation des entrées par thread

Par exemple, prenons une base de données de référence composée des patrons suivants : open write, open write close, write gettime getpid gettime, gettime getpid. On construit l'automate déterministe fini correspondant à ces patrons comme indiqué dans la figure 3.4. Cet exemple comprend 11 états numérotés de 1 à 10. Les états 1, 2, 3 et 4 sont des états finaux dont chacun correspond à un patron donné. Par exemple l'état 1 correspond au patron "open write", l'état 2 correspond au patron "open write close", etc.

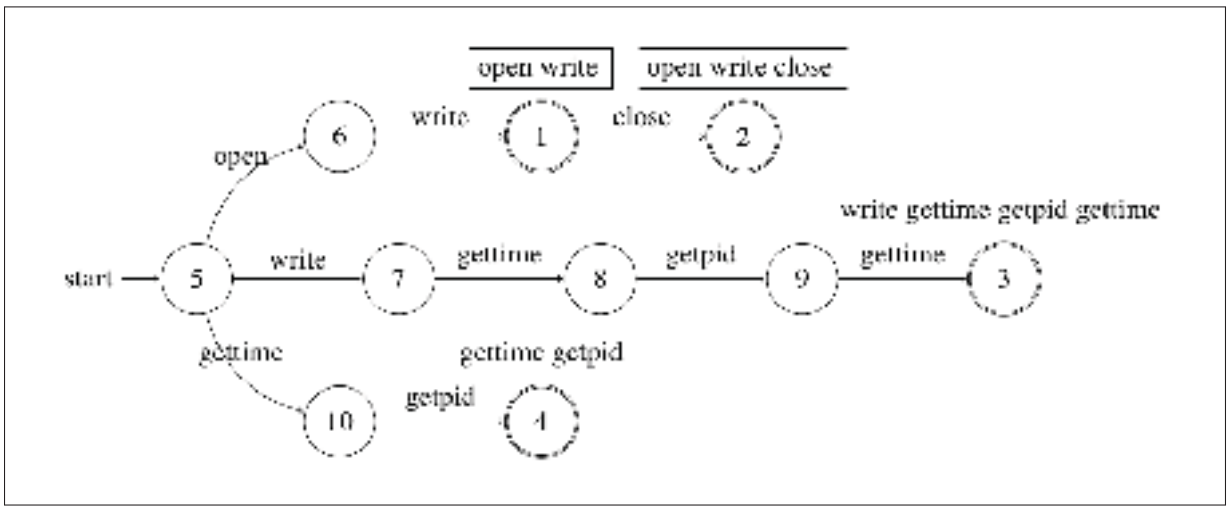


Figure 3.4 Exemple d'architecture d'automate déterministe fini de PFAC construit à partir de patrons d'appels système

On considère ensuite la trace d'exécution en entrée illustrée dans la figure 3.5. Chaque thread t_i commence la recherche de patrons malicieux depuis sa position de départ i. Par exemple, t_1 analyse la séquence "write gettime getpid gettime ..." et trouve une correspondance avec le 3ème patron "write gettime getpid gettime", il continue sa recherche au niveau du 5ème appel système de la séquence. Cependant, à ce niveau il n'y a plus de transitions valides. Le thread termine alors son traitement et reprend le traitement depuis sa position initiale précédente incrémentée du nombre total des threads du bloc. L'avantage d'un tel traitement réside dans le fait que la probabilité que les threads terminent le travail assez rapidement est élevée puisque chacun est responsable de trouver les patrons adéquats depuis leurs positions de départ et qu'ils terminent la recherche dès qu'on trouve une transition d'échec. Par exemple, dans la figure 3.5

le thread t_3 termine son travail dès la première position puisqu'il n'y a pas de transition valide à ce niveau. C'est le cas également pour les threads t_5, t_7 et t_8.

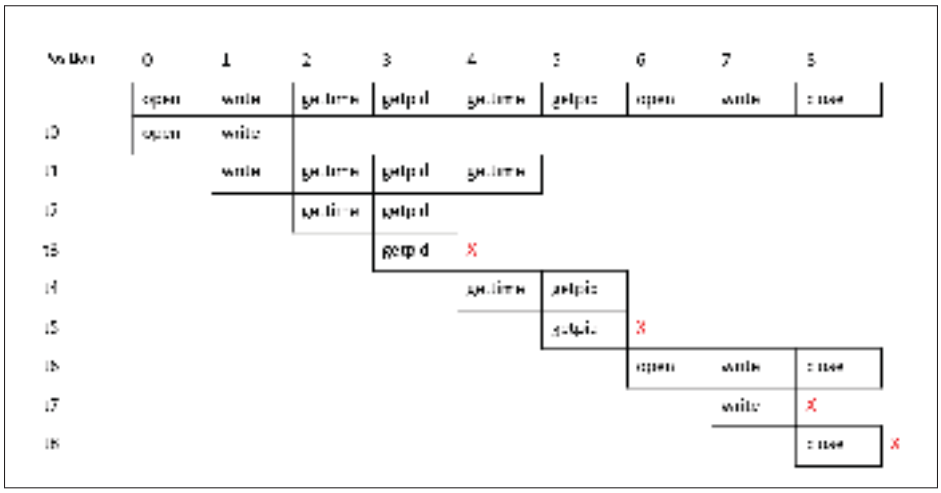


Figure 3.5 Analyse en parallèle avec PFAC d'une trace d'exécution

Une fois le traitement parallèle est terminé les résultats de la recherche seront envoyés à la CPU. Si on trouve une des signatures malicieuses de la base de données de référence dans l'une des traces d'exécution analysées, l'application relative à cette trace sera considérée malicieuse.

3.2.2 Le module de cryptographie

La cryptographie représente l'un des axes principaux des traitements raltives à la sécurité en général. Notre Framework intègre un module de cryptage et décryptage parallèle de données afin de protéger les données sensibles hébergées dans le téléphone comme les messages, les images, la liste des contacts, les notes, etc. L'algorithme parallèle de cryptographie que nous utilisons est le AES parallèle (PAES). Le choix pour cet algorithme de cryptographie s'est particulièrement porté pour sa large utilisation dans les systèmes mobiles ainsi que la complexité de calcul qu'il porte.

Le principe de ce module est le suivant. Afin de traiter toutes les données à crypter ou à décrypter, la CPU lance la requête d'exécution de plusieurs kernels sur la GPU. Comme l'indique

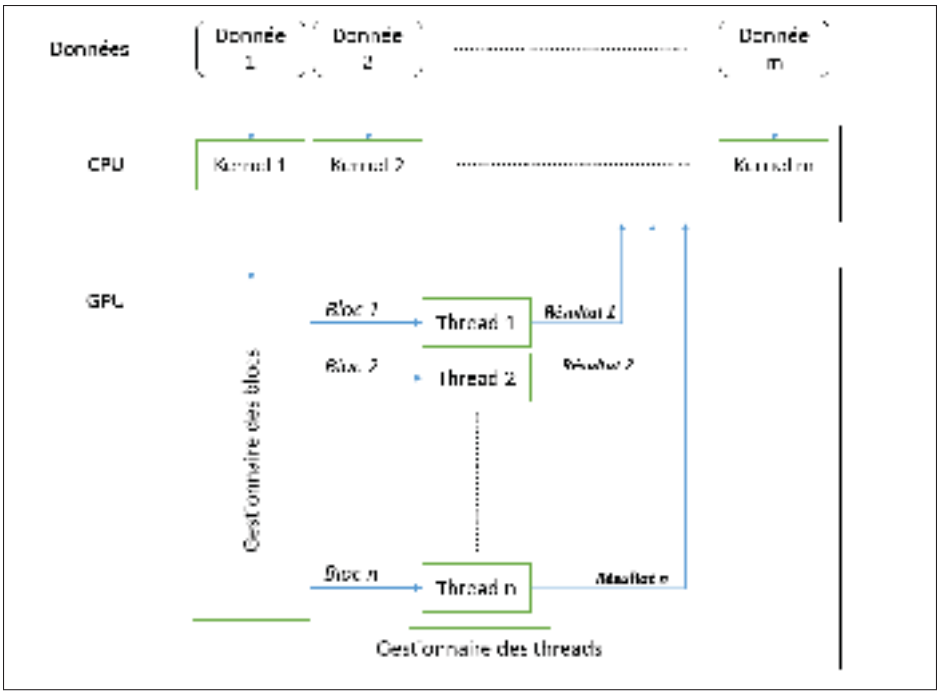


Figure 3.6 Le module de cryptographie parallèle

la figure 3.6, la GPU comporte trois principales composantes : le gestionnaire de l'exécution des threads, le gestionnaire des blocs et enfin les threads. La GPU prend en entrée le texte à traiter ainsi que la clé et les places dans la mémoire globale. Le texte est segmenté en blocs de 16 bytes chacun. Chaque thread prend en charge le traitement d'un bloc des données et envoie le résultat sous forme de texte crypté ou décrypté vers la CPU. Le texte final sera le résultat de la concaténation des différents blocs en ordre.

Les threads du même bloc OpenCL ont besoin d'un accès fréquent à la liste des clés générées. C'est pour cette raison que nous l'avons placé dans la mémoire globale de la GPU avec les données à traiter. Comme indiqué dans la figure 3.7, chaque thread ensuite exécute les différentes transformations relatives à l'algorithme de cryptographie AES à savoir la substitution des bytes *SubBytes*, les opérations de décalage *ShiftRows*, le mixage des colonnes *MixColumns* et enfin l'opération de combinaison des données avec les clés générées *AddRoundKey*.

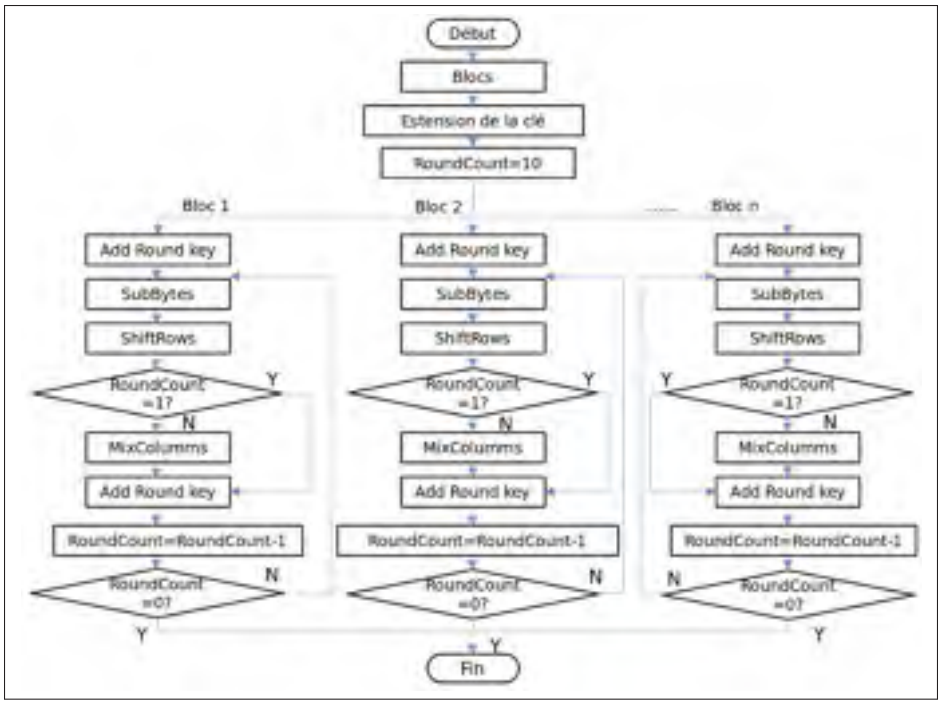


Figure 3.7 Le diagramme de l'algorithme PAES parallèle

3.2.3 Architecture hautement parallèle

Dans cette section nous allons décrire le module du cluster de notre architecture. Ce module offre une plateforme hautement parallèle sur deux niveaux : répartition parallèle des tâches sur les différents noeuds et l'exécution parallèle de ces taches au niveau de chaque noeud.

3.2.3.1 Le Cluster

Les téléphones intelligents sont des appareils à ressources limitées. Leur capacité de calcul et leur mémoire réduites limitent les traitements relatifs à la sécurité et les rendent plus critiques. L'architecture du cluster dans notre cas renforce la détection parallèle de malwares implémentée au sein du téléphone intelligent.

Le cluster que nous avons implémenté est de type Beowulf basé sur des cartes embarquées à ressources limitées (dans notre cas ce sont les cartes Parallella). Ce module offre des unités de calculs parallèles et distribués exécutant les mêmes algorithmes de détection de malwares

et de cryptographie discutés précédemment. En cas de non-disponibilité de ressources (CPU, GPU, mémoire, etc.) sur le téléphone intelligent, les traitements de détection et de cryptanalyse seront envoyés vers le cluster par le module d'envoi des données.

Le cluster que nous avons implémenté suit l'architecture du maître/esclave où le noeud maître reçoit les données à traiter et les envoie vers ses esclaves qui sont interconnectées via le réseau Ethernet. Le maître participe également dans l'exécution des traitements parallèle et effectue de plus les mêmes traitements que ses esclaves. Les données à traiter sont donc divisées équitablement sur les différents noeuds y compris le noeud maître. Cependant, au niveau des données du module de détection de malwares, on ajoute pour chaque bloc de données à envoyer vers les noeuds du cluster un segment dont la longueur est égale au plus long patron de la base de données des signatures afin de détecter les patrons qui sont dans la frontière de deux segments successifs. La gestion et la distribution des tâches au niveau du cluster se font avec la plateforme MPI comme c'est indiqué dans la figure 3.8. Il est à noter que la conception du module du cluster ainsi que son implémentation ont été faites en collaboration avec notre équipe de recherche.

3.2.3.2 Traitement parallèle au niveau des noeuds

Notre architecture du framework sur les téléphones mobiles a été portée et adaptée sur une plateforme hautement parallèle qui est la Parallella. La puissance de calcul parallèle de ces types de cartes et leurs prix réduits leur ont permis d'être une source de plus en plus attrayante pour le calcul à haute performance. Le coprocesseur Epiphany représente l'élément central du calcul parallèle relatif à notre architecture sur la carte. L'ensemble de la communication logicielle entre le système d'exploitation et le coprocesseur est géré par la librairie Epiphany SDK (ESDK). Cette librairie fournit l'ensemble des différentes fonctions permettant la communication et le chargement de code au niveau du coprocesseur. La plateforme OpenCL est fournie à travers la librairie COPRTHR qui offre des outils facilitant l'utilisation du calcul hétérogène visant à la fois les processeurs et les coprocesseurs.

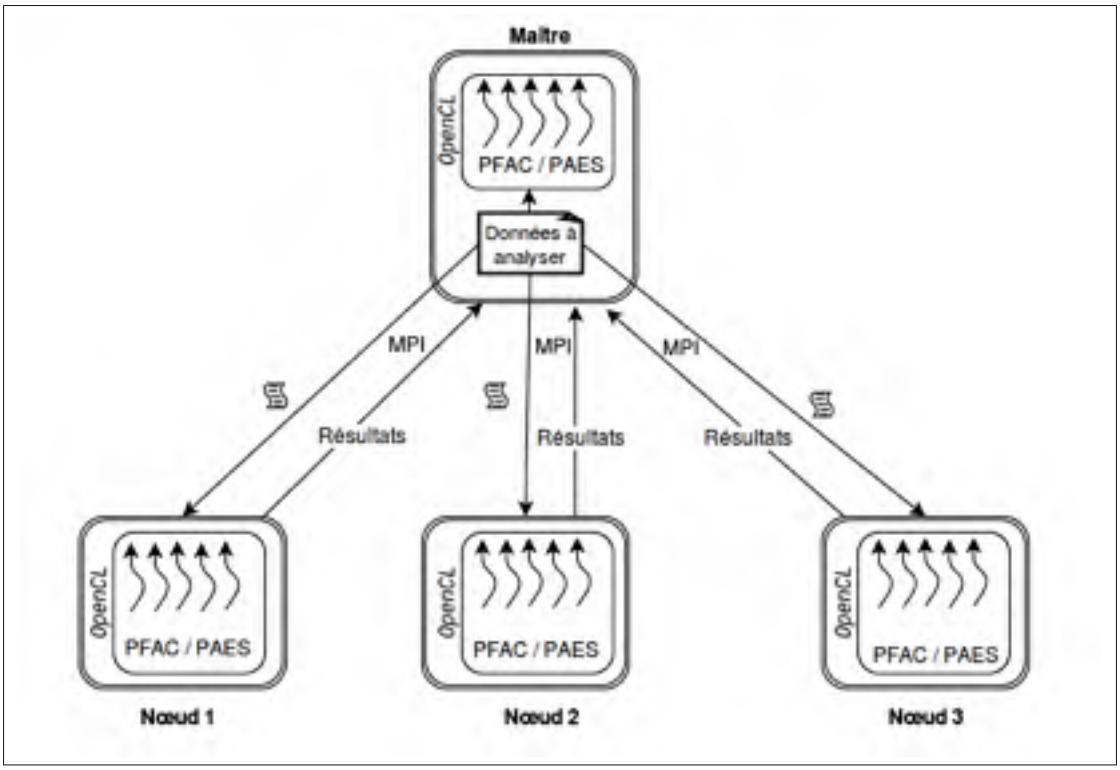


Figure 3.8 Architecture du cluster

Comme indiqué dans la figure 3.9, l'architecture générale portée sur la Parallella est presque la même que sur le téléphone mobile. Deux types d'algorithmes parallèles peuvent être exécutés sur le coprocesseur : PFAC pour la détection des malwares et PAES pour le cryptage et le décryptage des données. Une fois le traitement parallèle est terminé, les résultats seront envoyés vers le noeud maître.

3.3 Optimisations

Dans le but d'accélérer le traitement et optimiser le processus de détection de malware, une série d'optimisations sont introduites à notre architecture au niveau de la mémoire ainsi qu'au niveau du processus de traitement de données.

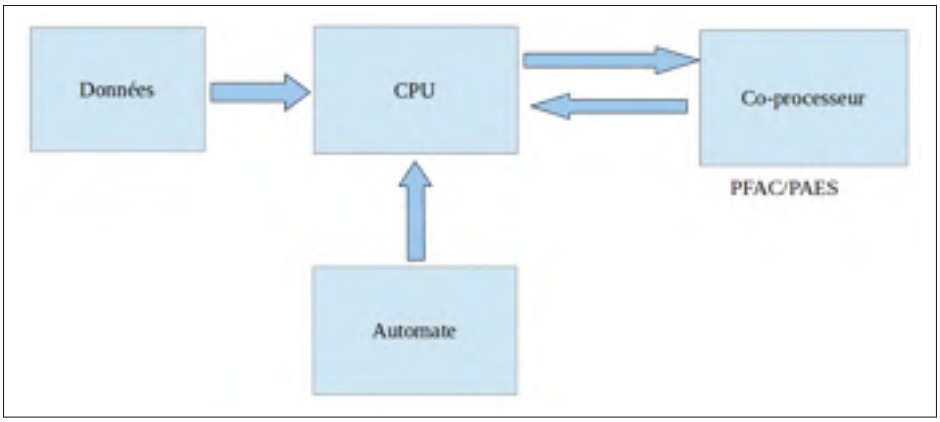


Figure 3.9 Architecture simplifiée sur la Parallella

3.3.1 Optimisation du stockage des données

Le stockage des structures d'automates dans un système embarqué (plus particulièrement les appareils mobiles) peut être problématique. En effet, le nombre de malwares est en perpétuelle évolution par la suite le nombre de signatures évolue de même. Ainsi la taille de l'automate devient de plus en plus importante. Comme nous sommes dans un milieu contraint ayant des ressources limitées il est indispensable d'appliquer des techniques de compactage de données afin d'optimiser le stockage des ressources nécessaires à la détection des malwares.

3.3.1.1 Élimination des transitions d'échec

La première technique de compactage des données que nous avons appliquée consiste à éliminer les transitions d'échec de l'automate comme l'indique la figure 3.10. En effet, avec l'algorithme parallèle de détection de patrons malicieux que nous utilisons, on peut ne pas prendre en considération ce type de transitions. Dans le processus de recherche, il n'est plus nécessaire de revenir d'un pas en arrière dans l'automate avec ces transitions puisque chaque thread commence la recherche depuis sa position initiale et termine sa recherche locale dès qu'il ne trouve pas une transition valide depuis sa position courante. La table des transitions d'échec est alors éliminée ce qui permet de réduire l'espace de stockage de la structure d'automate utilisée.

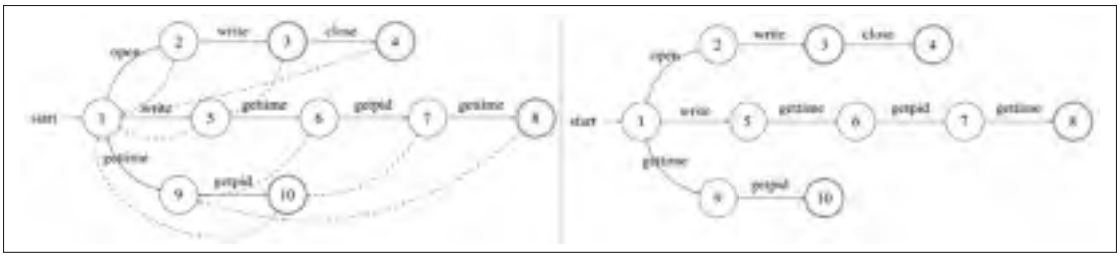


Figure 3.10 Elimination des transitions d'échec de la structure d'automate

3.3.1.2 Élimination de la table des états finaux

La table des états finaux permet d'indiquer si un état correspond à la fin d'un patron donné. Cependant, on peut éliminer cette table et connaitre si un état donné est final ou normal à l'aide d'un processus de renumérotation des états dans l'automate. En effet, après avoir construit l'automate à partir de la base de données des patrons, on peut renuméroter les états finaux de 1 jusqu'à m ou m est le nombre total des patrons utilisés et les autres états de m+1 jusqu'au nombre total des états. Ainsi pour connaitre si un état est final ou normal il suffit de vérifier s'il est inférieur ou supérieur au nombre total des patrons. Avec cette technique on peut éliminer la table des états finaux et réduire encore l'espace requis pour le stockage de notre structure d'automate.

3.3.1.3 Compactage de l'automate avec P3FSM

Pour réduire la taille de l'automate, nous avons exploité la technique de compactage qu'offre l'algorithme P3FSM. Cet algorithme permet de réduire considérablement la taille de l'automate à travers l'élaboration de deux tables descriptives de l'automate en question. La première est "la table des codes" contenant les différents états codés et la seconde est "la table caractère / groupe" qui regroupe principalement les informations concernant l'alphabet utilisé. Une étude de l'efficacité de cet algorithme va être élaborée dans le chapitre suivant ainsi qu'une comparaison au niveau du taux de compactage des données avec l'algorithme de hachage parfait décrit dans le chapitre de l'état de l'art.

3.3.2 Partitionnement optimal des blocs d'unités de traitement parallèle

Comme nous utilisons OpenCL dans notre architecture pour l'exécution parallèle, il faut spécifier la configuration optimale de la taille des unités de traitement qui encapsulent les threads. Cette étape a pour but de maximiser le débit d'exécution dans l'unité de traitement parallèle, à savoir la GPU ou le coprocesseur Epiphany. Ceci est élaboré à travers la spécification de nombre total des threads ainsi que de la taille des unités de traitement optimal qui offre un maximum de débit d'exécution.

3.3.3 Adopter un scénario optimal pour la gestion d'envoi des fichiers à analyser

Au niveau du module de détection de notre architecture, nous analysons le profil d'exécution d'applications stockées dans différents fichiers. Ces fichiers vont être envoyés vers le buffer de l'unité de traitement parallèle pour détecter la présence éventuelle d'un malware. Afin d'assurer une performance maximale de notre architecture, il est indispensable de bien choisir la politique d'envoi des fichiers à analyser surtout que le coût de transfert des données vers les GPU mobile est élevé et engendre un temps de latence important.

Pour cet effet, nous avons expérimenté trois scénarios différents pour l'envoi des traces à analyser afin de déduire le meilleur scénario offrant le débit d'exécution le plus élevé. Ces scénarios ainsi que les différents résultats vont être détaillés dans le chapitre suivant.

3.3.4 Utilisation optimale des types de mémoires des unités de traitement parallèle

Pour maximiser la performance du traitement parallèle, il est indispensable de bien exploiter les différents types de mémoires que possèdent en général les périphériques offrant le traitement parallèle.

En effet, en ce qui concerne les GPUs, ils possèdent principalement trois types de mémoires (voir figure 3.11). Premièrement, la mémoire globale qui est accessible par n'importe quel thread de la GPU. Elle n'est pas une mémoire cache et il faut attendre plusieurs cycles pour y

accéder. En d'autres termes, le temps de latence pour ce type de mémoire est plus important que les autres, ce qui laisse un multiprocesseur inactif pendant ce temps. Le deuxième type de mémoire qu'on trouve dans une GPU est la mémoire locale. Cette mémoire est, à l'instar de la mémoire globale, n'est pas une mémoire cache, mais possède un temps de latence moins élevé. Cette mémoire a une capacité de stockage moins importante que la mémoire globale et n'est utilisée que pour certaines variables qui sont partagées par un même bloc de threads. Il faut donc choisir d'y placer les données qui seront exploitées par un même bloc et dont l'accès et fréquent. Quant à la mémoire constante, la lecture ne coûte généralement qu'un cycle puisque c'est une mémoire cache. Cependant l'espace de stockage est moins important que les deux autres types de mémoires. Ainsi, pour assurer une performance maximale, il est recommandé de placer les données qu'utilisent tous les threads et dont l'accès est fréquent, non pas dans la mémoire globale, mais dans la mémoire constante. Ceci permet de minimiser le temps de latence et augmenter ainsi le débit du traitement parallèle du système.

Dans le chapitre suivant, nous allons expérimenter plusieurs configurations pour le placement des données dans les différents types de mémoires de la GPU mobile pour déterminer celle qui offre le meilleur débit.

3.4 Construction de la base des signatures

Dans notre architecture, nous utilisons une base de signatures de malwares comme référence, extraits à partir de la liste des appels système émis par ces derniers. Nous allons décrire dans cette section le processus de construction de la base des signatures des applications Android malicieuses et qui correspond à la première étape de la partie prétraitement de notre architecture. Cette approche est proposée par (Lin *et al.*, 2013b). Elle se fait principalement avec trois étapes : extraction des séquences d'appels systèmes par application, extraction des séquences communes et enfin le filtrage des séquences malicieuses.

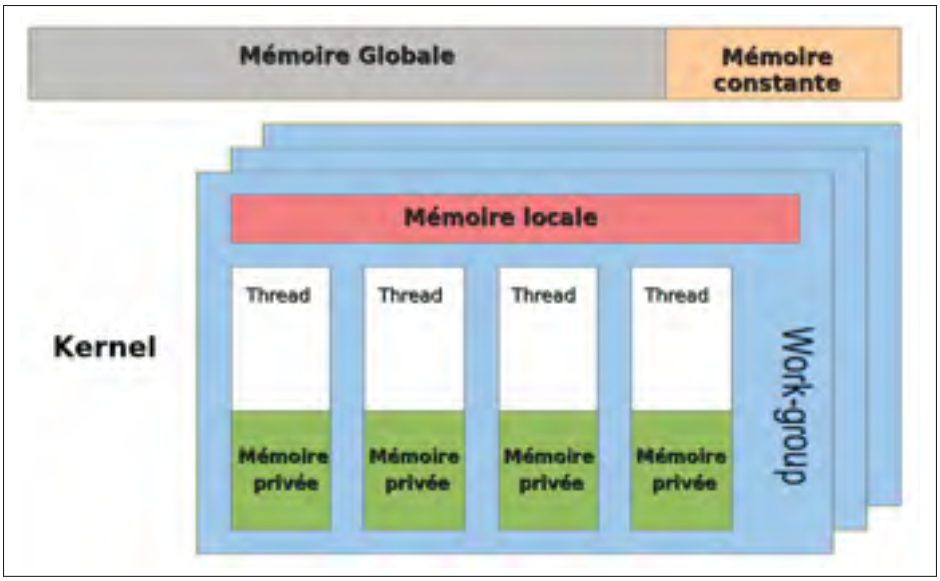


Figure 3.11 Architecture mémoire des GPUs mobiles avec OpenCL
Tirée de OpenCL (2015)

3.4.1 Extraction des séquences d'appels systèmes

Dans cette étape nous avons besoin de deux types d'applications : des applications malicieuses groupées par famille de malwares (M) ainsi que des applications normales (N). Dans une première étape, nous exécutons séquentiellement les applications normales sur un système Android et nous enregistrons pour chaque application (i) les séquences d'appels systèmes (BSi). Nous utilisons pour cet effet l'outil *Strace* qui permet de tracer les appels systèmes par application avec leurs paramètres. On filtre par la suite ces appels systèmes pour éliminer les paramètres et ne grader que les séquences d'appels systèmes bruts. Après avoir enregistré toutes les BSi nous obtenons l'ensemble B = {BSi | 1<= i <=|N| }.

Nous considérons par la suite la liste des applications malicieuses groupées par famille de malwares (M). Nous exécutons chaque application Mi d'une même famille de malware dans un système Android et nous examinons la liste des threads (j) émis par chaque processus parent pour ces applications. Nous enregistrons par la suite les séquences des appels systèmes (Sij) pour chaque thread de l'application malicieuse. La structure d'arborescence des threads pour

chaque application également enregistrée. Après avoir enregistré toutes les Si nous obtenons l'ensemble S = {Si | 1<= i <=|M| }.

3.4.2 Extraction des séquences communes

Le but de cette étape est de collecter les plus longues séquences d'appels systèmes qui sont communes entre les applications d'une même famille de malwares pour indiquer la présence possible d'un comportement malicieux. Par exemple, on considère les deux séquences suivantes : « write open mmap open socket close gettime write » et « gettime write write open open mmap open socket close write ». La plus longue séquence commune extraite sera « mmap open socket close ». Comme les applications Android exécutent plusieurs threads par application, les séquences d'appel système Si relatives à chaque thread seront généralement longues et le nombre de comparaisons entre les traces d'exécution pour extraire les séquences communes sera énorme. Pour pallier à ce problème (Lin *et al.*, 2013b) proposent un mécanisme de comparaison multithreads hiérarchique pour extraire d'une manière efficace les séquences communes malicieuses. Ils affirment qu'un comportement malicieux activé par un même code malicieux et hébergé dans des applications différentes apparaitra dans le même niveau dans la structure d'arborescence des threads créés par ces applications. Ainsi, pour extraire les séquences qui sont potentiellement malicieuses on ne compare que les traces d'exécution des threads du même niveau et qui appartiennent à la même famille de malwares.

3.4.3 Filtrage des séquences malicieuses

La liste des séquences communes d'appels systèmes ainsi obtenue ne peut pas être directement utilisée, car on peut trouver quelques patrons dans des applications normales. Le processus de filtrage est alors indispensable pour ne garder que les patrons d'appels systèmes exprimant des comportements malicieux. Pour ce faire, nous utilisons le théorème de Bayes pour calculer la probabilité qu'une séquence donnée soit malicieuse. Ce théorème est exprimé par la formule suivante :

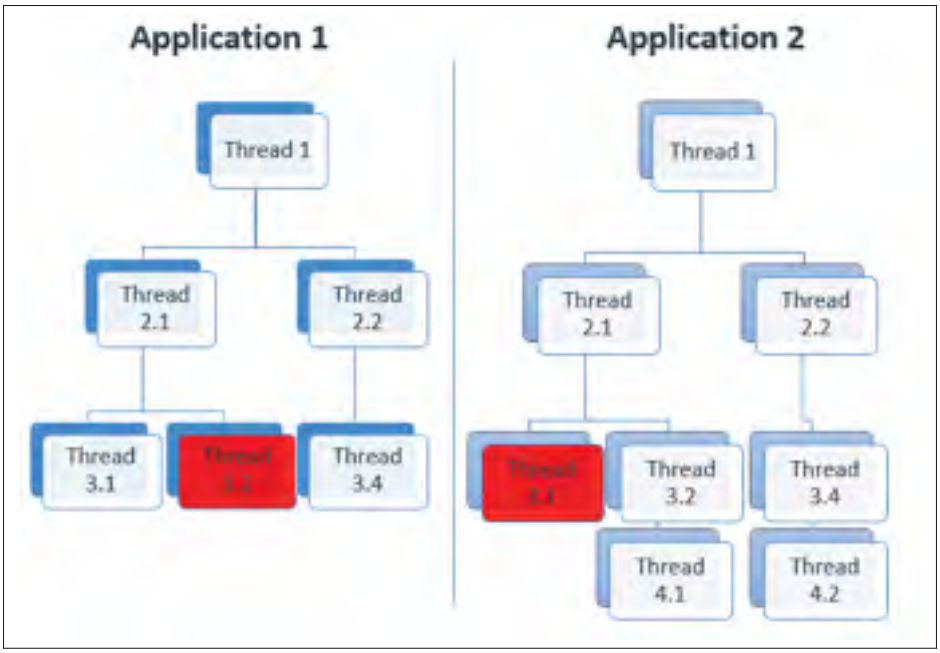


Figure 3.12 Arborescences des threads de deux applications d'une même famille de malware

$$\frac{P(SCi|M)x(M)}{P(SCi|M)x(M)+P(SCi|B)xP(B)} \tag{3.1}$$

Où :

P(N) *: la probabilité que l'application soit normale.*

P(M) *: la probabilité que l'application soit malicieuse.*

P(SCi\|N) *: la probabilité que la séquence commune SCi apparaisse dans des applications normales.*

P(SCi\|M) *: la probabilité que la séquence commune SCi apparaisse dans des applications malicieuses.*

Dans notre cas, nous travaillons avec les séquences communes dont la probabilité qu'elles soient malicieuses est de 100%. En d'autres termes, nous gardons les patrons communs d'appels systèmes qui sont présents dans les applications malicieuses et non pas dans le profil d'exécution des applications normales. Pour terminer, la figure suivante illustre bien les trois

étapes de construction de la base de signatures comportementales des malwares que nous allons utiliser par la suite pour le traitement et la détection parallèle.

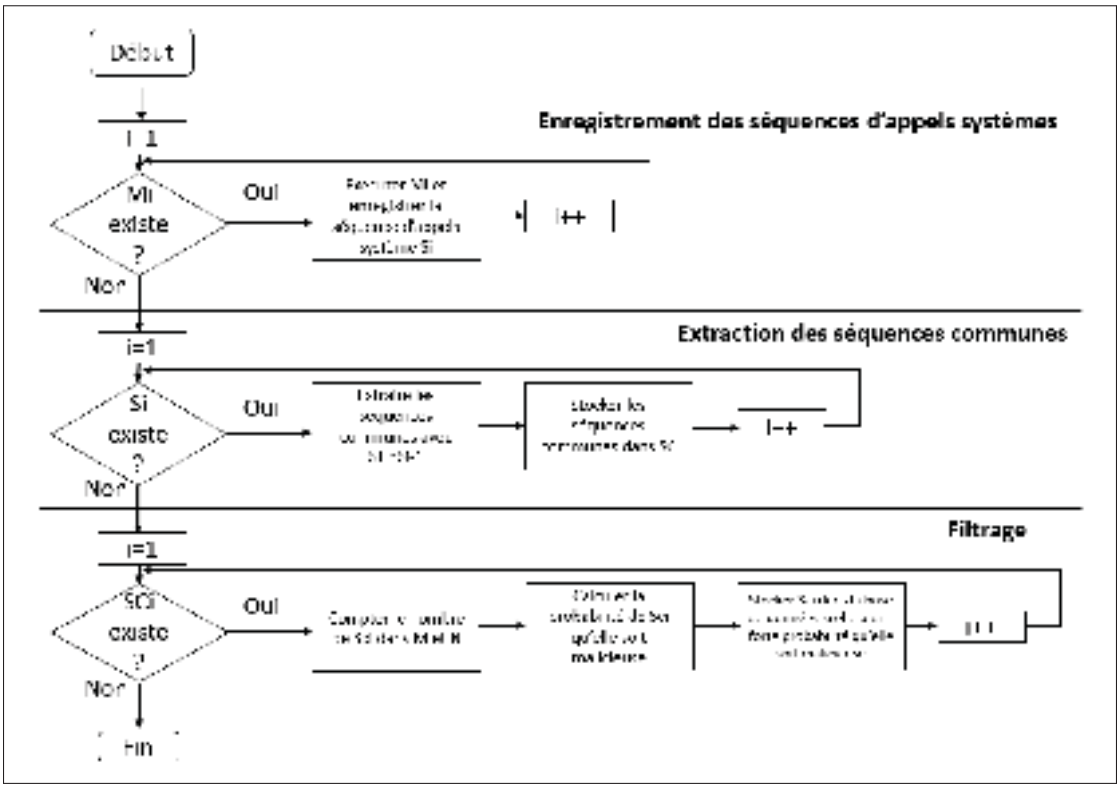


Figure 3.13 Les étapes de construction de la base des signatures

3.5 Conclusion

Dans ce chapitre nous avons décrit notre architecture d'accélération de traitements de la sécurité mobile avec des systèmes à ressources limitées (téléphones mobiles et le cluster des cartes Parallella). Nous avons également décrit les différentes optimisations faites pour être conformes aux contraintes de mémoire et de capacités de calcul réduites de ces systèmes embarqués. Une étude expérimentale et une validation de l'approche proposée seront élaborées dans le chapitre suivant.

CHAPITRE 4

ÉTUDE EXPÉRIMENTALE ET VALIDATION

4.1 Introduction

Ce chapitre va s'intéresser à l'étude expérimentale qui a été menée pour valider l'architecture parallèle proposée et ses différents modules. Une première partie de ce chapitre va présenter les différentes expérimentations élaborées avec la GPU mobile pour déterminer la meilleure configuration accélérant le traitement de la détection de malwares et de cryptographie sur un téléphone mobile. Dans une deuxième partie de ce chapitre, nous allons décrire les différentes expérimentations pour accélérer le traitement parallèle au niveau d'un noeud du cluster pour aboutir enfin aux tests de performance finaux pour le cluster en général.

4.2 Expérimentations avec la GPU mobile

Nous avons implémenté notre architecture sur un téléphone mobile Sony Xperia Z muni d'un microprocesseur Qualcomm Snapdragon 600, quadricœur avec une fréquence égale à 1.7 GHz ainsi que d'une GPU mobile Qualcomm Adreno 320 munie de 16 cœurs avec une fréquence d'horloge de 400 MHz. La version d'Android que nous avons utilisé est la 4.4.1.

Plusieurs expérimentations ont été établies afin d'optimiser le traitement parallèle. Dans ce qui suit, nous allons décrire la liste des expérimentations tout en analysant les résultats obtenus.

Nous commençons alors par la définition des métriques suivantes :

$$Débit\ d'exécution = \frac{Tailles\ des\ données\ analysées}{Temps\ de\ traitement} \tag{4.1}$$

$$Accélération = \frac{Temps\ de\ traitement\ séquentiel}{Temps\ de\ traitement\ parallèle} \tag{4.2}$$

4.2.1 Détermination de la taille optimale du groupe de travail local dans la GPU

Comme nous l'avons détaillé auparavant, les threads exécutés sur la GPU sont regroupés dans des ensembles de même taille appelés groupes de travail. Il est nécessaire de bien déterminer la taille de ces ensembles de threads avant l'exécution des kernels dans la GPU afin de maximiser la performance du calcul parallèle. Nous faisons donc varier la taille du groupe de travail local et pour chaque configuration, nous déterminons le débit d'exécution.

Comme l'indique la figure 4.1, le meilleur débit est obtenu en exécutant 16 threads par groupe. Nous remarquons également qu'au-delà de cette valeur, plus on augmente le nombre de threads par groupe de travail plus le débit diminue. Ceci est lié étroitement à la fréquence d'horloge des cœurs de la GPU qui est limitée. En outre, l'exécution d'un nombre excessif de threads par cœur va engendrer un overhead qui va ralentir l'exécution et diminuer le débit d'exécution. Dans ce qui suit, nous allons donc garder cette configuration pour les expérimentations ultérieures puisqu'elle nous donne le meilleur débit.

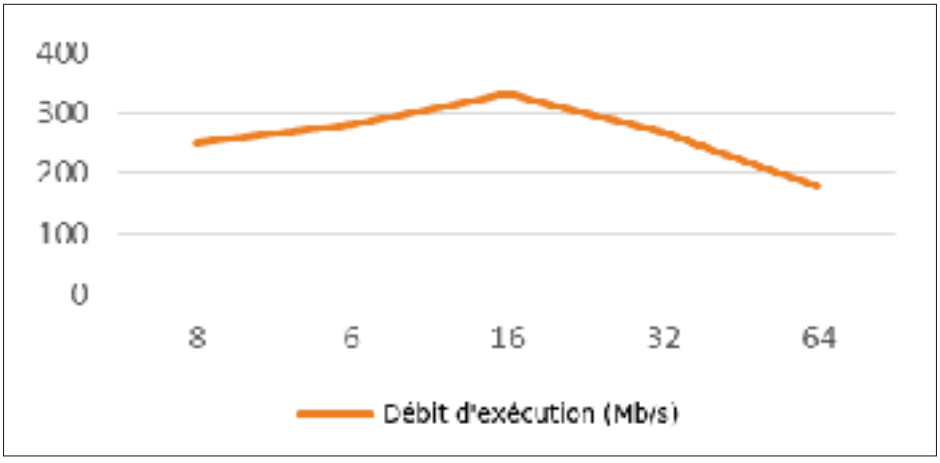


Figure 4.1 Evaluation de la taille optimale du groupe de travail local

4.2.2 Détermination du placement optimal des données dans les différents types de mémoires de la GPU

Dans cette section, nous allons expérimenter différentes configurations pour le placement des données au niveau de la GPU mobile afin d'en déduire celle qui offre le meilleur débit. Pour cela, nous avons considéré six configurations comme indiqué dans le tableau 4.1. Les données mises en considération sont : la table des transitions, le tampon des entrées (contient les données en entrée à analyser) et le tampon des résultats du traitement.

Dans la configuration 1, nous avons mis toutes les données dans la mémoire globale et négligé l'exploitation des autres types de mémoires. Dans la configuration 2, nous avons gardé la table des transitions ainsi que le tampon des résultats dans la mémoire globale de la GPU et nous avons déplacé les données à analyser dans la mémoire locale. Dans la configuration 3, nous avons déplacé le tampon des entées vers la mémoire constante de la GPU qui est plus rapide en accès, mais de taille réduite (seulement 64 KB). Comme ce type de mémoire est réduit, il nous est impossible d'y mettre toute la table de transitions. C'est pour cette raison que nous avons divisé cette dernière en deux parties dans les configurations qui suivent. Dans la configuration 4, la première partie de la table des transitions est placée dans la mémoire constante, quant au reste, il est mis dans la mémoire globale avec le tampon des résultats. Dans la configuration 5, nous avons déplacé le tampon des données en entrée vers la mémoire globale et gardé la même configuration que précédemment pour les autres données. Dans la dernière configuration, nous avons mis le tampon des données en entrée dans la mémoire constante et la première partie de la table des transitions dans la mémoire locale du GPU. Le reste des données sont placées dans la mémoire globale.

Nous avons exécuté les configurations une à une sur l'Xperia Z et mesuré à chaque fois le débit d'exécution. La figure 4.2 résume les résultats obtenus. Le meilleur débit est obtenu avec la 4ème configuration où les données en entrée sont placées dans la mémoire locale et la table des transitions dans les mémoires constante et globale. En effet, si nous comparons les configurations 4 et 5, le fait de déplacer le tampon des données en entrée vers la mémoire globale engendre un temps d'accès plus important à ces données (surtout que l'accès à ce tampon est

Tableau 4.1 Les différentes configurations pour le placement des données dans les différentes mémoires de la GPU

Configuration	Mémoire globale	Mémoire constante	Mémoire locale
Configuration 1	Table des transitions Tampon des entrées Tampon des résultats	-	-
Configuration 2	Table des transitions Tampon des résultats	-	Tampon des entrées
Configuration 3	Table des transitions Tampon des résultats	Tampon des entrées	-
Configuration 4	Table des transitions P2 Tampon des résultats	Table des transitions P1	Tampon des entrées
Configuration 5	Table des transitions P2 Tampon des résultats Tampon des entrées	Table des transitions P1	-
Configuration 6	Table des transitions P2 Tampon des résultats	Tampon des entrées	Table des transitions P1

fréquent). De plus, en comparant les configurations 4 et 2 nous remarquons que le fait de placer la première partie de la table des transitions dans la mémoire constante améliore la performance du traitement puisque l'accès à la première partie de la table des transitions est fréquent et que la vitesse d'accès à ce type de mémoire est très rapide. Ceci a donné une amélioration dans la performance d'environ 19% par rapport à la deuxième configuration. Ainsi nous avons gardé la quatrième configuration pour le framework ainsi que pour les expérimentations qui suivent.

4.2.3 Accélération

Dans cette section nous allons comparer la performance de notre algorithme de détection parallèle par rapport à la version séquentielle. A partir de la figure 4.3, nous pouvons remarquer que toutes les configurations qui utilisent le calcul parallèle sur la GPU (de 1 à 6) sont plus

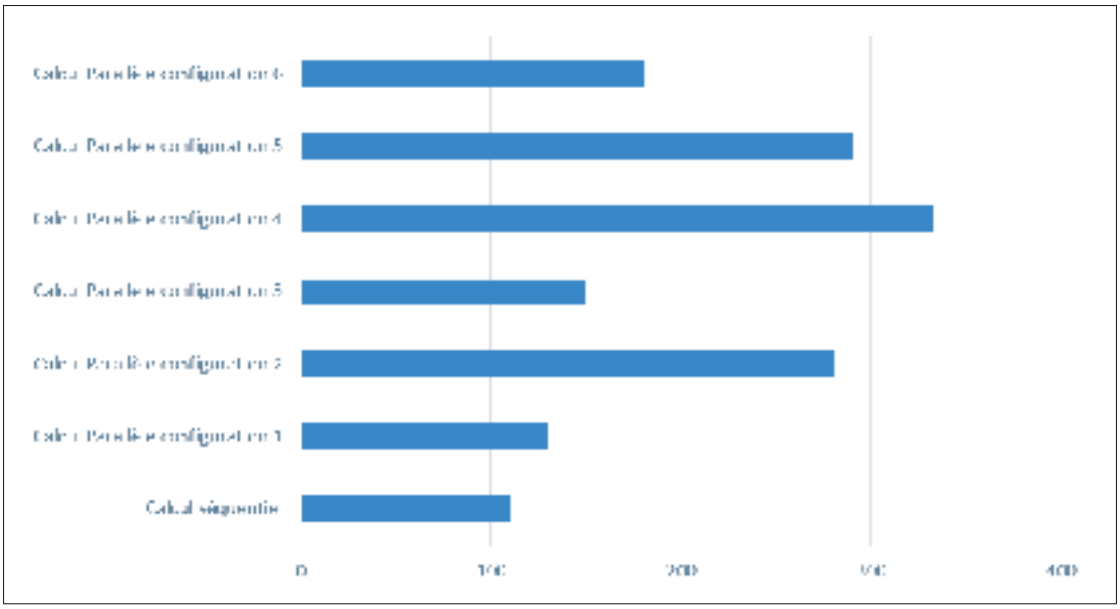


Figure 4.2 Débit d'exécution en (MB/s) des différentes configurations de mémoires

rapides en terme de débit d'exécution que la version séquentielle exécutée sur la CPU du téléphone mobile. Une accélération de 3 fois par rapport à la version séquentielle est obtenue avec l'utilisation de la configuration 4 détaillée précédemment.

Nous avons de même implémenté une version parallèle de l'algorithme PFAC sur la CPU mobile avec le paradigme Pthreads et comparé cette implémentation avec notre version parallèle sur la GPU mobile tout en faisant varier le nombre de threads qui sont en cours d'exécution. Pour la version sur la CPU nous remarquons que le meilleur débit est obtenu en exécutant 64 threads en parallèle. Cependant au delà de 80 threads, le débit commence à diminuer considérablement vu que la CPU exécute une charge de plus en plus lourde. Avec 128 threads, le système se plante et l'exécution est interrompue. En examinant les résultats expérimentaux, nous remarquons que le débit d'exécution du PFAC sur la GPU mobile est nettement meilleur qu'avec la CPU mobile. Ceci nous ramène à conclure que la GPU mobile offre une unité de calcul puissante pour le traitement parallèle et peut être considérée comme un bon candidat pour l'accélération de la détection de malwares sur les téléphones mobiles.

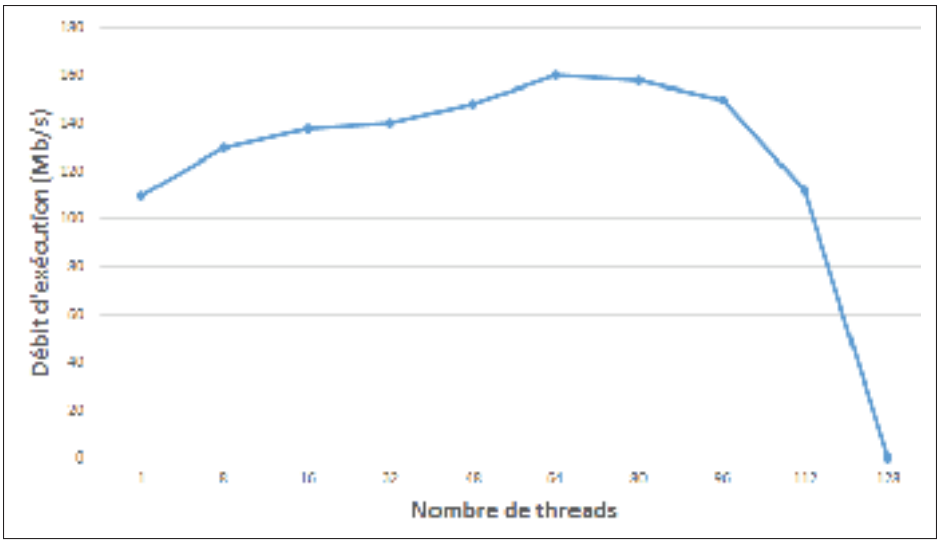


Figure 4.3 Débit d'exécution en (MB/s) de PFAC sur une CPU mobile

4.2.4 Politique d'analyse des fichiers de traces d'exécution des applications

Le transfert des données entre la CPU et la GPU est considéré comme l'une des contraintes majeures qui ralentissent le temps d'exécution du système. La minimisation du temps de latence engendré par ces transferts permettra d'améliorer la performance de calcul et avoir de meilleurs résultats.

A. Description des scénarios

Dans cette section nous allons étudier différents scénarios pour l'envoi des fichiers de traces d'exécution des différentes applications à analyser vers la GPU dans le but de déduire le scénario le plus optimal en termes de débit d'exécution.

Scénario 1 : on considère un tampon de données en entrées de taille fixe qui sera alloué à un seul fichier à la fois contenant les appels système d'une application donnée. L'envoi des fichiers vers le tampon se fait d'une façon séquentielle et la recherche des patrons malicieux à partir de ce tampon se fait d'une façon parallèle.

Scénario 2 : on considère dans ce type de scénarios un tampon des entrées de taille fixe et vers lequel on envoie les fichiers à analyser un par un. Si à un moment donné le tampon n'est pas totalement alloué, on envoie le fichier des traces d'exécution suivant à analyser.

Scénario 3 : on considère dans ce type de scénarios un tampon des entrées de taille fixe à plusieurs entées. En d'autres termes, plusieurs fichiers à analyser vont être envoyés à la fois vers le tampon de la GPU qui sera segmenté. Chaque segment est dédié aux traces d'un fichier donné. L'analyse des données va être exécutée en parallèle comme pour les autres scénarios.

B. Détermination du nombre de segments optimal pour le troisième scénario

Dans cette section nous allons étudier le nombre optimal d'applications que peuvent être analysées en parallèle qui sera égal au nombre de segments du tampon des données en entrée de la GPU mobile. Pour cela nous avons exécuté 100 applications sur le système Android 4.4.1 pendant trois minutes et collecté les appels systèmes émis par ces applications. Ces applications sont variées (jeux, média, communication, etc.) et leurs traces d'exécution sont de tailles différentes. Nous considérons un tampon pour les entrées segmenté en 5, 10, 15 et 20 morceaux. Chaque segment est dédié aux traces d'une application à analyser. Les threads de la GPU sont distribués équitablement pour chaque morceau. De plus, si nous arrivons à la fin d'un fichier dans le processus d'analyse et il reste encore d'autres à traiter, nous réduisons le nombre des segments et nous divisons de nouveau le tampon d'une façon équitable. D'après les résultats expérimentaux illustrés dans la figure 4.4, nous remarquons qu'en moyenne, le nombre optimal des applications à scanner en parallèle est égal à 10. Ainsi dans ce qui suit nous segmentons le tampon des entrées en 10 morceaux pour le 3ème scénario et nous comparons ensuite la performance du framework selon les trois scénarios.

C. Comparaison de la performance des scénarios

Afin d'étudier l'efficacité des trois scénarios, nous avons considéré différentes tailles du tampon du flux des données en entrée à analyser. Comme nous pouvons le voir dans la figure 4.5,

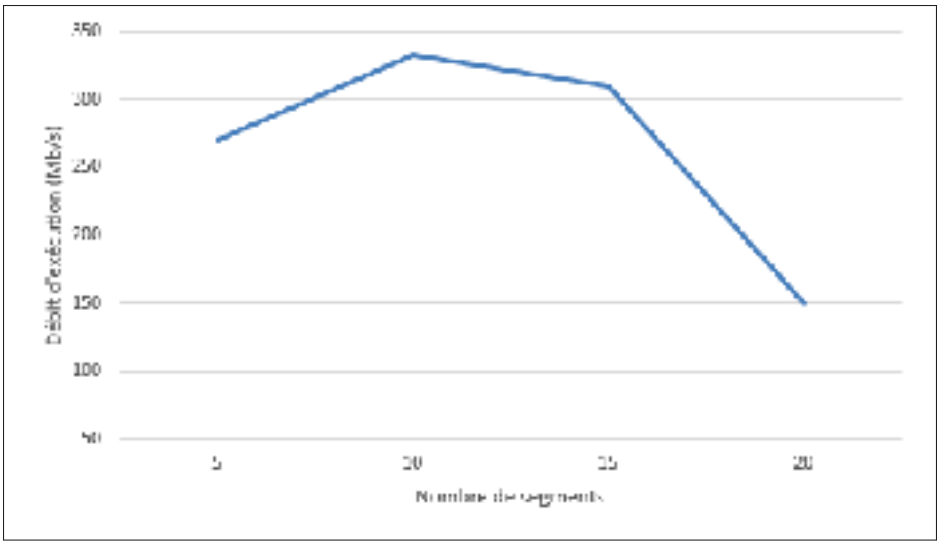


Figure 4.4 Étude de la variation du nombre de segments du tampon des entrées dans le scénario 3

le troisième scénario offre toujours le meilleur débit. En effet, pour le premier scénario, le tampon d'entrée de la GPU mobile n'est pas pleinement exploité. Il y a donc un gaspillage des ressources allouées ce qui affecte alors le débit d'exécution.

En outre, si nous considérons d'une part les deux premiers scénarios, plus de transferts de données de la mémoire de la CPU vers la GPU sont nécessaires pour traiter toutes les traces d'exécution à analyser. Par conséquent, le temps de traitement devient plus important avec ces deux premiers scénarios, car le coût de transfert des données vers la GPU est plus élevé comparé à la troisième configuration. D'autre part, les entrées multiples dans le tampon du troisième scénario offrent moins de temps de latence pour l'exécution sur la GPU en raison de son efficacité dans l'exploitation de l'allocation de tampon d'entrée. D'après la variation de la taille des données à analyser, nous remarquons également que plus le tampon d'entrée est grand, plus le temps total d'exécution est plus rapide, et ce pour les trois configurations. Il est d'ailleurs recommandé de minimiser les transactions entre les mémoires de la CPU et de la GPU mobile à travers l'envoi de plus gros paquets de données au lieu d'envoyer plusieurs paquets dont la taille est réduite. Enfin, il faut noter que le débit d'exécution du module de

détection de malwares sur la GPU reste dominé par les transferts de données entre la CPU et la GPU qui est égale à environ 65% de la durée totale de traitement.

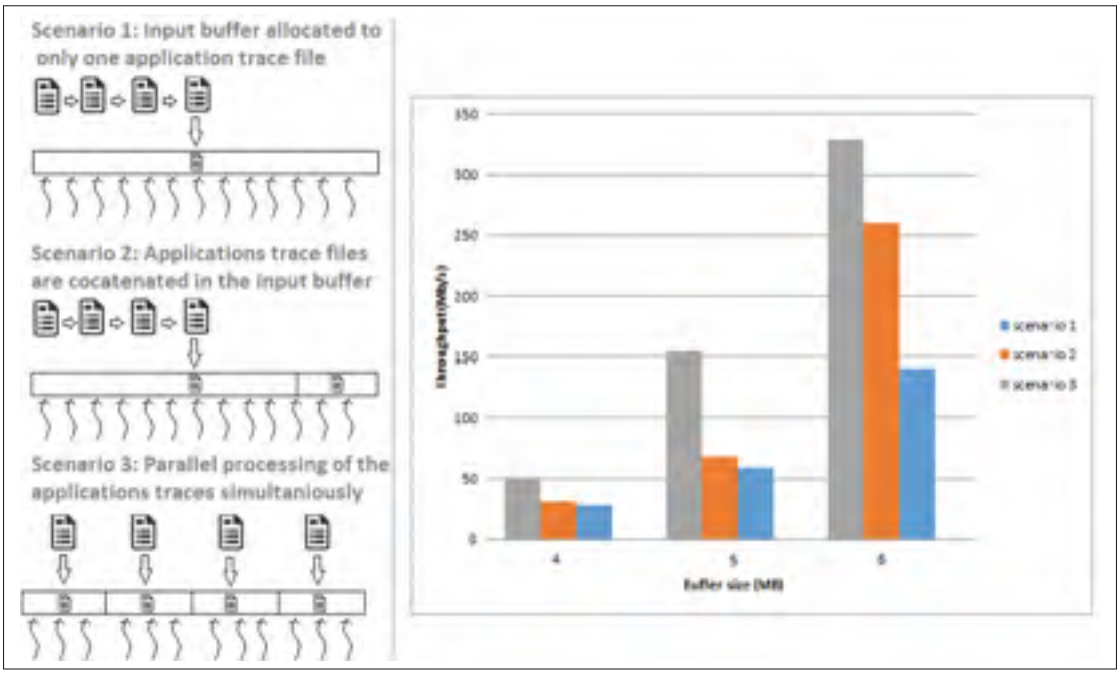


Figure 4.5 Comparaison de la performance des scénarios

4.2.5 Comparaison des techniques de compactage de données

Stocker une structure d'automate dans un téléphone mobile peut être problématique surtout avec l'évolution rapide du nombre de malwares et les capacités de stockage réduites offertes par les smartphones. Une technique de compactage de données efficace est alors indispensable pour assurer un meilleur niveau d'efficacité en maximisant le nombre des signatures malicieuses avec lequel nous pouvons travailler.

Dans ce but, nous avons implémenté trois techniques de compactage de données durant la phase du prétraitement qui sont le Failureless-AC (Aho-Corasick sans les transitions d'échecs), P3FSM et le hachage parfait. Nous avons appliqué chaque algorithme sur différentes tailles d'automates générées à partir d'un nombre de signatures malicieuses différentes. Le but est de

déterminer la technique de compactage la plus efficace qui maximise le nombre de patrons malicieux que peut supporter la mémoire de la GPU. Le tableau 4.2 résume les résultats obtenus.

Comme nous le pouvons remarquer P3FSM minimise le plus l'espace mémoire requis pour stocker l'automate. Le taux de compactage offert par un tel algorithme est aux alentours de 10 fois par rapport à Failureless-AC. Quant à l'algorithme de hachage parfait, il offre un taux de compactage égal à 3x par rapport au même algorithme. Nous remarquons également que l'algorithme du hachage parfait offre un taux de compactage de données de moins au moins important avec l'évolution du nombre de signatures à considérer. Ceci est dû au fait que la matrice de transitions devient de plus en plus dense. Donc la réduction de la matrice en une seule ligne devient de plus en plus complexe et moins avantageuse. Ainsi nous avons gardé dans notre framework la technique de compactage de P3FSM puisqu'elle réduit le plus la mémoire requise pour stocker l'automate dans la mémoire de la GPU. Cet algorithme permet également de considérer et de stocker un nombre plus important de patrons malicieux. Finalement, en termes de temps d'exécution nous notons que l'algorithme P3FSM et beaucoup plus rapide que le hachage parfait qui devient de plus en plus lent si la matrice transition devient plus dense.

Tableau 4.2 Capacités de mémoire requises pour différentes techniques de compactage de données

Nombre de patrons	Failureless-AC (KB)	Hachage parfait (KB)	P3FSM (KB)
2000	67677	22674	8922
2200	74398	25611	9234
10000	678937	254417	50765
16000	806554	389327	60432
17600	809321	412745	74380

4.2.6 Accélération de l'algorithme de cryptographie parallèle

Afin d'étudier la performance de l'algorithme de cryptographie parallèle PAES sur un téléphone mobile, nous avons exécuté cet algorithme sur la GPU avec différentes tailles de don-

nées. Nous avons calculé le temps d'exécution pour chaque ensemble de données et le comparé à l'exécution séquentielle sur la CPU mobile.

D'après les résultats expérimentaux illustrés dans la figure 4.6, nous remarquons que l'utilisation de la GPU pour le PAES devient plus intéressante avec une taille de données à crypter supérieure à 2.4 MB. Ceci est dû au temps de latence causé par le chargement des données vers la GPU qui ralentit l'exécution par rapport au CPU. Un taux d'accélération de 1.3x est obtenu avec l'utilisation du calcul parallèle sur la GPU mobile par rapport à la version séquentielle. Ainsi, nous pouvons noter que l'utilisation de la GPU mobile permet d'accélérer le traitement de l'algorithme AES et peut être utilisée pour augmenter la performance de tel traitement.

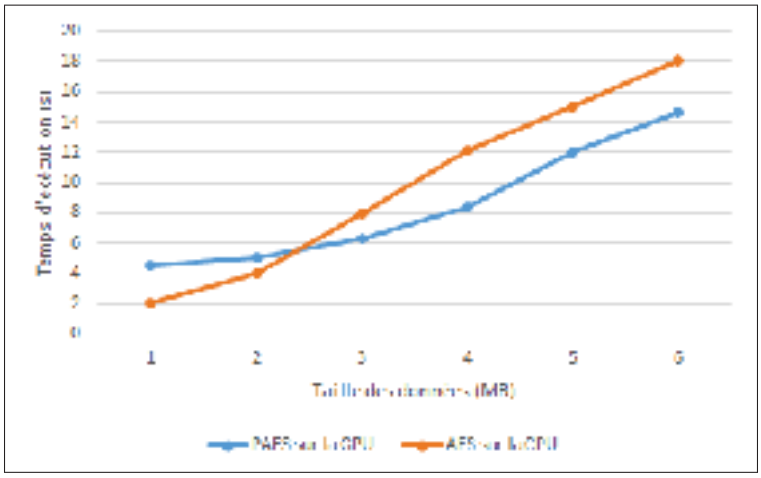


Figure 4.6 Comparaison du temps d'exécution du PAES sur la GPU mobile et le AES séquentiel

4.2.7 Consommation d'énergie

La contrainte majeure pour la programmation d'applications mobiles est la consommation d'énergie. Dans cette section, nous allons discuter la consommation d'énergie de notre framework sur le téléphone mobile et le comparer à différentes applications. Dans ce but, nous avons exécuté les algorithmes parallèles PFAC ainsi que le PAES et mesuré le pourcentage de la consommation de la batterie. Les résultats obtenus et regroupés dans le tableau 4.3 affirment

que la consommation d'énergie de notre framework est bonne comparé aux autres applications et ne dépasse pas les 0.26%.

Tableau 4.3 Pourcentage de consommation d'énergie des algorithmes parallèles implémentés comparés à d'autres applications

Application	**Pourcentage d'utilisation de la batterie**
Avast Mobile security	10%
Système Android	6.9%
Kernel (android os)	4.6%
Google Play	1%
Starbucks	2%
Hangouts	0.2%
Skype	3%
Google+	0.3%
PFAC	0.1%
PAES	0.16%

4.3 Expérimentations avec la Parallella

Avant la phase d'implémentation de l'architecture de cluster de notre solution et afin de garantir un meilleur niveau de performance, il est indispensable de maximiser la performance de calcul au niveau de chaque noeud. La performance des deux algorithmes PFAC et PAES a été alors étudiée et comparée aux versions séquentielles pour les cartes Parallella.

4.3.1 Performances du PFAC

Pour la partie de détection de malwares nous avons essayé de maximiser la performance de l'algorithme de correspondance de patron PFAC en faisant varier la taille locale et globale du groupe de travail. En utilisant le coprocesseur Epiphany à 16 coeurs de la carte Parallella, le nombre maximal des threads est égal à 16. Quant à la taille locale maximale du groupe de travail sur le coprocesseur est égal à 3 threads/coeur.

Nous avons calculé le débit d'exécution relatif à l'algorithme tout en faisant varier la taille locale et globale du groupe de travail. Les résultats expérimentaux sont illustrés dans la figure 4.7. Il faut noter qu'au niveau de l'axe des abscisses, les couples illustrés dans le diagramme reflètent respectivement la taille globale et la taille locale du groupe de travail avec lequel nous travaillons. Le meilleur débit d'exécution obtenu est égal à 3.1 Gb/s avec 8 coeurs exécutant 2 threads chacun.

D'après les résultats expérimentaux illustrés dans la figure 4.7 nous remarquons que plus le nombre de threads est grand, plus le débit d'exécution s'améliore. De plus, nous remarquons que plus on maximise le nombre de threads par coeur plus le débit est meilleur. Par exemple, si nous comparons la performance des deux derniers couples (16,1) et (16,2) nous remarquons qu'il est préférable de travailler avec 8 coeurs qui exécutent chacun 2 threads, qu'avec 16 coeurs exécutant chacun un seul thread. Ainsi dans les expérimentations qui vont suivre nous allons garder cette configuration puisqu'elle nous fournit le meilleur débit d'exécution.

La performance maximale de PFAC obtenue a été également comparée à la version séquentielle de l'algorithme. Une accélération de 5x a été obtenue avec l'utilisation de la version parallèle sur le coprocesseur Epiphany. Cependant, un temps de latence égale à environ 50% du temps d'exécution est généré avec l'algorithme parallèle à cause du délai d'initialisation de la plateforme OpenCL sur le coprocesseur et le chargement des données.

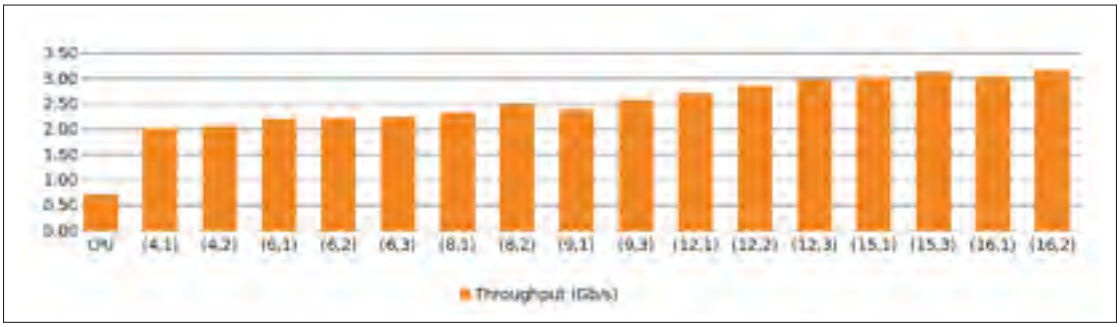


Figure 4.7 Débit d'exécution de PFAC en fonction de la taille du groupe de travail local

4.3.2 Performances du PAES

Pour le module de cryptographie implémenté dans la carte Parallella, nous avons étudié sa performance en exécutant l'algorithme de cryptographie parallèle PEAS sur le coprocesseur Epiphany tout en faisant varier la taille des données à crypter ou à décrypter (de 1MB à 8MB). Le temps d'exécution pour chaque entrée de données a été enregistré et comparé à la version séquentielle. D'après la figure 4.8, nous remarquons que grâce à l'utilisation du PAES sur le coprocesseur Epiphany une accélération de 1.5x est obtenue comparée à la version séquentielle. Ces résultats sont prometteurs pour plus d'accélération au niveau du cluster formé par les cartes Parallella.

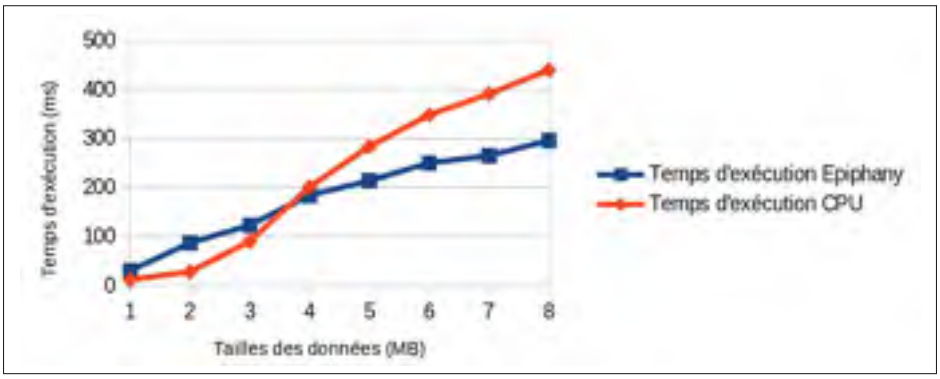


Figure 4.8 Temps d'exécution du PAES et du AES séquentiel

4.4 Expérimentations avec le cluster

Pour étudier la performance de l'architecture du cluster de notre plateforme, nous avons élaboré une série d'expérimentations concernant la taille du cluster ainsi que la taille des données à traiter. Puisque nous disposons de quatre cartes Parallella, nous avons travaillé avec un cluster qui est formé d'au maximum quatre noeuds.

4.4.1 Performances du PFAC dans le cluster

Le but de l'expérimentation suivante est de vérifier le potentiel de la mise à l'échelle de notre cluster ainsi que sa performance si nous augmentons le nombre de noeuds. Pour ce faire nous avons considéré des tailles différentes de traces d'exécutions d'applications à scanner. Ces données sont distribuées par le noeud maître vers les noeuds esclaves. Nous augmentons à chaque fois le nombre total de noeuds du cluster et nous enregistrons le débit d'exécution total. Les résultats expérimentaux sont illustrés dans la figure 4.9. L'augmentation de la taille du cluster permet d'avoir un débit d'exécution de plus en plus important. Nous remarquons alors que l'ajout de noeuds dans le cluster permet d'une part d'accélérer le traitement parallèle du module de détection de malwares et de traiter d'autre part plus efficacement une plus grande quantité de données.

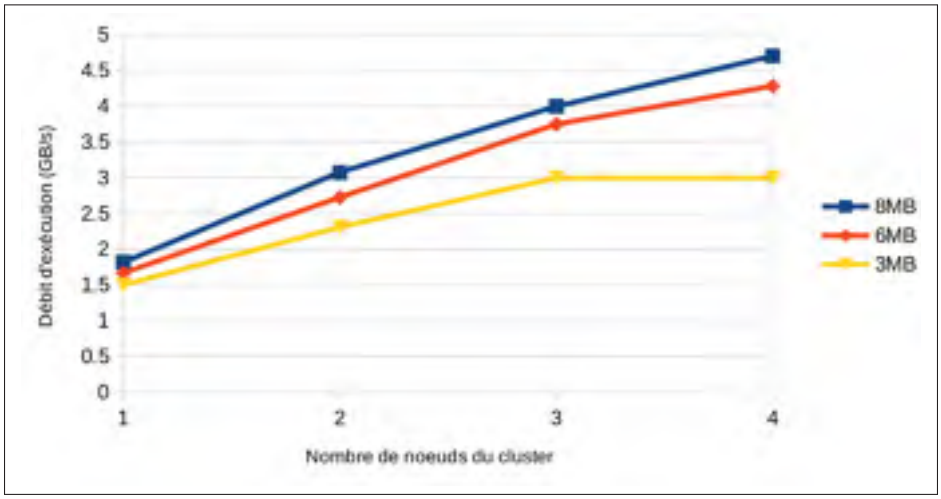


Figure 4.9 Débit d'exécution du PFAC dans le cluster en fonction du nombre de noeuds et la taille des données à traiter

4.4.2 Performances du PAES dans le cluster

La même expérimentation que précédemment a été élaborée pour tester la performance de l'algorithme de cryptographie parallèle PAES sur notre architecture de cluster. D'après la figure 4.10 l'utilisation de 4 noeuds au niveau du cluster a donné une accélération de 3x comparé au

traitement parallèle sur une seule carte et une accélération de 5x comparé à la version séquentielle.

Ainsi, nous pouvons conclure que l'utilisation de l'architecture de cluster des cartes Parallella améliore d'une part la performance de la détection de malwares et accélère d'autre part le module de cryptographie proposé.

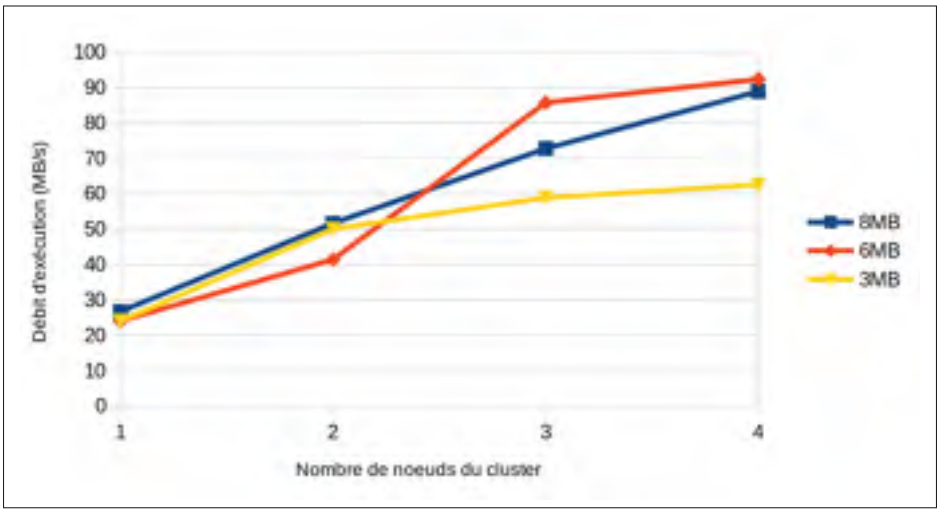


Figure 4.10 Débit d'exécution du PAES dans le cluster en fonction du nombre de noeuds et la taille des données à traiter

4.5 Construction de la base des signatures et précision de détection

Dans cette section nous allons aborder la phase de la construction de la base des signatures de malwares et l'évaluation de la précision de détection de l'approche de détection utilisée.

4.5.1 Construction de la base des signatures

Pour construire la base des signatures de mlawares nous avons considéré 10 familles listées dans le tableau 4.4. Chaque famille contient plusieurs applications qui hébergent le même malware. Nous avons exécuté chaque application malicieuse sur un émulateur Android pour assurer un environnement de test fiable. Ensuite, pour chaque application nous avons extrait la

séquence d'appels systèmes relative à chaque thread lancé par l'application ainsi que l'arborescence des threads et ce avec l'outil *strace*. Un exemple de trace d'exécution est illustré par la figure 4.11.

Tableau 4.4 Le nombre d'applications malicieuses traitées dans la phase d'apprentissage ainsi que celle utilisées dans la phase de détection groupées par famille

Famille de malwares	**Taille de l'échantillon dans la phase d'apprentissage**	**Taille de l'échantillon évalué**
ADRD	11	10
AnserverBot	94	93
ASroot	4	4
BaseBridge	61	61
BeanBot	4	4
DroidDream	8	8
DroidKungFu	48	48
Geinimi	35	34
GoldDream	24	23
KMin	26	26
Total	**315**	**313**

Figure 4.11 Exemple de profil d'exécution d'une application malicieuse

Les paramètres des appels systèmes sont ensuite éliminés pour ne garder que les traces brutes. Après la phase d'extraction des plus longues séquences communes des traces de la même famille de malware, la phase de filtrage commence. Pour cet effet, nous avons exécuté 313 applications normales dans un téléphone Sony Xperia Z1 et enregistré la liste des appels systèmes de chaque application pour construire un profil d'exécution global pour les applications non malicieuses. Toute séquence commune extraite précédemment et qui existe à la fois dans les traces des applications malicieuses ainsi que les traces normales est éliminée de la base de données des signatures malicieuses. Le processus de filtrage a permis une réduction importante de la base des signatures, ce qui est détaillé dans le tableau 4.5. D'après les résultats expérimentaux nous pouvons constater que plus le nombre d'applications malicieuses par famille est important plus les séquences communes obtenues avant le processus de filtrage sont précises. L'évaluation de la précision de détection de notre base de données de signatures malicieuses sera abordée dans la section suivante.

Tableau 4.5 Statistiques des pourcentages de réduction des signatures dans le processus de filtrage

Famille de malwares	**Pourcentage de réduction du nombre des signatures après le filtrage**
ADRD	63%
AnserverBot	19%
ASroot	58%
BaseBridge	11%
BeanBot	49%
DroidDream	45%
DroidKungFu	8%
Geinimi	26%
GoldDream	56%
KMin	32%

4.5.2 Précision de détection

Dans cette section nous allons étudier la précision de détection de notre framework. Nous allons alors définir les terminologies suivantes :

- **TP** (*True positif*) : le taux d'applications normales correctement détectées comme non malicieuses ;
- **TN** (*True Negative*) : le taux d'applications malicieuses correctement détectées comme malicieuses ;
- **Nm** : le nombre d'applications malicieuses ;
- **Nb** : le nombre d'applications normales ;
- **DA** (*Detection accuracy*) : la précision de détection calculée par la formule suivante :

$$DA = \frac{TP * Nm + TN * Nb}{Nm + Nb} x100\% \tag{4.3}$$

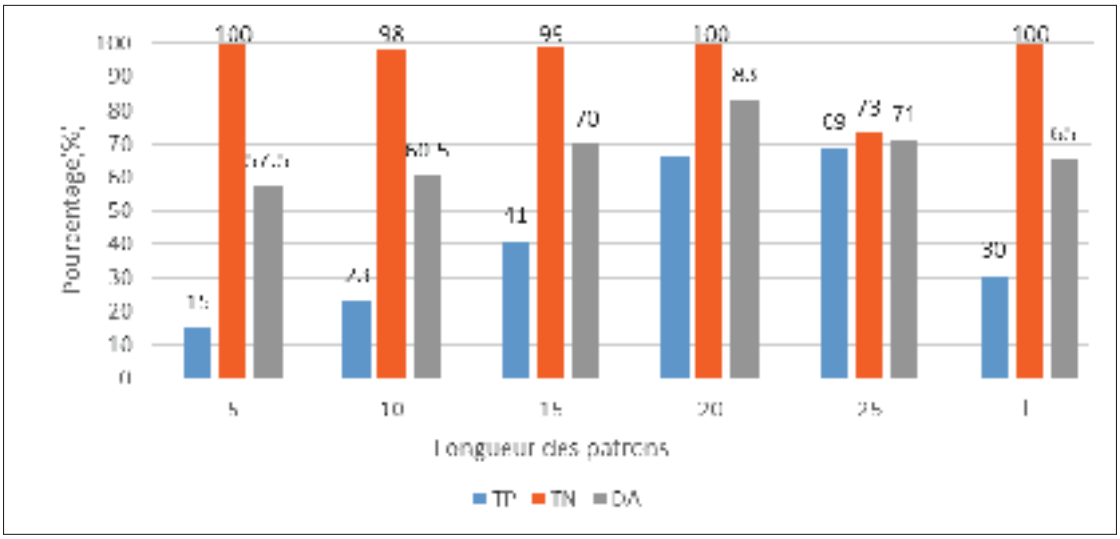


Figure 4.12 Etude de l'impact de la variation de la longueur des patrons sur la précision de détection

Afin de mesurer la précision de détection, nous avons joué sur la longueur des patrons des signatures malicieuses extraites et les avons comparées à la base des signatures de longueur variable. Nous avons alors varié la taille des signatures et mesuré à chaque fois le taux de TP, TN et la précision de détection. D'après les résultats expérimentaux illustrés dans la figure 4.12 nous pouvons constater que la meilleure précision de détection est obtenue avec des signatures de longueur 20 avec une précision de détection égale à 83%. Les signatures de taille variables que nous avons obtenues, souffrent d'un taux de TP réduit puisque la base contient toujours

des séquences d'appels systèmes non malicieuses. D'après les résultats obtenus, nous pouvons conclure que les signatures de taille 20 reflètent le mieux le comportement malicieux des applications Android grâce à la précision de détection qu'elle offre. Ainsi nous avons gardé ce type de signatures dans notre base de données de référence avec laquelle nous avons travaillé.

4.6 Discussions

Pour valider notre architecture proposée, une série d'expérimentations a été élaborée. Pour commencer, au niveau de la méthode de détection de malwares nous avons opté pour une technique de détection basée sur les signatures comportementales de malwares. Cette technique permet de détecter une éventuelle attaque en cours d'exécution et permet à partir d'une seule signature de détecter toute une famille de malware. La précision de détection offerte par une telle technique est relativement bonne. Cependant ces résultats peuvent être améliorés à travers l'apprentissage de comportements de plus d'applications normales ainsi que malicieuses afin de mieux filtrer les patrons non malicieux de la base de données.

Au niveau du module de détection parallèle sur le téléphone mobile, la détermination de la taille du groupe de travail locale et globale est indispensable. Cependant, la taille optimale du groupe de travail local n'est pas la même pour toutes les GPUs. Cette opération doit alors être adaptée avec chaque GPU mobile utilisée. Ensuite, une utilisation judicieuse des différents types de mémoires de la GPU a été menée pour optimiser le processus de détection parallèle. D'ailleurs plusieurs types d'optimisations au niveau de la mémoire des GPUs standards ne sont pas supportés par les GPUs mobiles ce qui rend l'optimisation du traitement plus difficile. Nous avons donc ensuite regardé comment peut-on avoir un scénario optimal pour l'envoi des données à analyser par le module de détection vers la GPU pour minimiser le temps de latence du module de détection. En effet, réduire le trafic entre la GPU et la mémoire est l'une des principales techniques pour augmenter la performance du calcul parallèle au niveau de la GPU. Il est vrai que la segmentation du tampon des entrées a donné un meilleur résultat par rapport aux autres configurations, cependant ceci ne reste valide que pour les traces d'exécution de tailles différentes et relativement importantes.

Concernant le module de cryptographie sur le téléphone mobile, son intégration dans notre architecture avait pour but de prouver qu'il est possible d'utiliser les GPUs mobiles pour accélérer les traitements cryptographiques. Les résultats pour le décryptage n'ont pas été indiqués dans le rapport puisque ça donne les mêmes performances que celle de cryptage.

Il est à noter que dans notre plateforme nous avons supposé l'existence de moniteur des ressources au niveau du téléphone et qui sera responsable de l'envoi des données vers une architecture hautement parallèle, qui est le cluster. Parmi les robustesses de la plateforme OpenCL est le fait qu'il soit portable. Nous avons donc porté notre architecture sur les cartes Parallela et ciblé le coprocesseur Epiphany pour accélérer le traitement parallèle. L'architecture mémoire du coprocesseur diffère de celle des GPUs. Il nous était donc impossible de faire des optimisations au niveau du placement des données ou même exploiter la mémoire locale des coeurs du coprocesseur puisqu'elle est réduite (34Kb pour le code et les données du kernel). Toutes les données sont donc placées dans la mémoire globale du système. D'après les résultats expérimentaux, les cartes Parallella offrent une plateforme efficace pour les traitements hautement parallèles vu leur coût réduit et les performances de calculs offertes. Ces résultats peuvent être améliorés à travers l'intégration d'un moniteur au niveau du noeud maître pour équilibrer la charge de calcul au niveau du cluster et assurer un plus haut niveau de fiabilité du système.

CONCLUSION

L'accélération des traitements de la sécurité mobile est devenue une préoccupation de plus en plus importante vu la croissance exponentielle des attaques ciblant ces plateformes d'une part, et le développement de la capacité de calcul parallèle dans ce type de systèmes d'autre part. Il est indispensable de protéger les informations sensibles au sein des téléphones mobiles à travers l'implantation de systèmes de détection de malwares ainsi que le chiffrement des données dans le but de maintenir un plus haut niveau de sécurité. Cependant, ce type de traitements est lourd surtout qu'il s'agit d'un cadre de systèmes à ressources limitées en termes de mémoire, de batterie et de capacité de calcul.

Dans ce travail de recherche, nous nous sommes intéressés à l'accélération des traitements de la sécurité mobile par le calcul parallèle. Nous avons alors proposé une architecture optimisée pour l'accélération de la détection de malwares sur les téléphones mobiles ainsi que les traitements relatifs à la cryptographie par le calcul parallèle. Cette architecture est composée de deux principaux éléments : la détection de malwares et la cryptographie parallèles sur les GPUs mobiles ainsi que leurs extensions avec l'implémentation de l'architecture de cluster.

Tout d'abord, nous avons commencé par une revue de littérature sur les principaux axes de recherche pour mieux cibler nos contributions. Particulièrement, au niveau de l'étude des techniques d'accélération des algorithmes de pattern matching, nous n'avons trouvé aucun travail qui s'intéresse à ce type de problèmes sur les GPUs mobiles. Les techniques proposées qui existent s'intéressent plutôt aux GPUs standards et pour la plupart elles ne sont pas supportées par l'environnement mobile. Ceci a rendu notre phase de recherche plus difficile. Cette difficulté a été surmontée grâce à un filtrage adéquat des techniques d'accélération et la proposition d'autres optimisations adaptées à l'environnement des systèmes à ressources limitées.

Ensuite au niveau de la GPU mobile, nous avons rencontré des difficultés pour accéder à ce périphérique avec OpenCL. En effet, Google a récemment bloqué l'accès direct aux APIs OpenCL dans certains systèmes Android pour favoriser l'utilisation de son nouvel outil RenderScript qui n'a pas eu beaucoup de succès. Cette difficulté a été surmontée à travers le test de

plusieurs types de plateformes mobiles pour trouver enfin celle contenant le pilote de OpenCL qui demeure accessible.

Ensuite, au niveau du module de détection sur le téléphone mobile, nous avons eu des difficultés concernant le type des signatures malicieuses sur lesquelles nous allons travailler. Les signatures de hachage des malwares Android ne sont pas publiques et pour la plupart elles ne sont pas adéquates pour notre environnement. Ce problème a été résolu à travers la construction de notre propre base de signatures de malwares comportementales en se basant sur (Lin *et al.*, 2013b).

Nombreuses sont les contributions de notre travail de recherche. Pour commencer, nous avons ciblé dans une première étape les GPUs mobiles non pas à des fins de traitements graphiques, comme elles sont communément exploitées, mais plutôt à des fins de sécurité. Nous avons utilisé différentes techniques d'accélération de traitement parallèle sur les GPUs mobiles. Une exploitation efficace du placement des données dans les différents types de mémoires de la GPU a permis d'accélérer le traitement parallèle de notre architecture. La proposition d'un scénario optimal pour l'envoi des fichiers à analyser vers la GPU et qui est basé sur les entrées multiples a permis également d'accélérer le traitement parallèle et de minimiser le temps de latence dû au transfert de données entre la CPU et la GPU.

La deuxième contribution de ce mémoire est l'utilisation et l'adaptation des techniques de compactage des données relatives aux structures d'automates pour remédier à la contrainte de la mémoire réduite offerte par les systèmes embarqués. Plus particulièrement, l'utilisation de P3FSM a donné un taux de compactage important (10x) ce qui nous a permis de travailler avec un nombre plus important de signatures malicieuses.

La troisième contribution de notre travail de recherche est l'utilisation d'une technique de détection basée sur l'analyse comportementale d'applications malicieuses offrant une bonne précision de détection. Une base de données de signatures malicieuses a été également élaborée pour construire le modèle de référence. Parmi les points forts de notre architecture proposée,

c'est que le même traitement de détection reste toujours valable pour d'autres types de données comme les appels API, les permissions, les signatures de hachage de malwares, etc.

La dernière contribution de notre travail de recherche est l'extension de notre plateforme vers l'architecture de cluster dans le cas de non-disponibilité des ressources dans le téléphone mobile. Le coût réduit des clusters formés à partir des systèmes embarqués ainsi que la performance de plus en plus évoluée et offerte par ce type de systèmes ont motivé notre choix pour ce type d'architecture qui représente une source attrayante pour les calculs parallèles traditionnellement implémentés dans des serveurs puissants et distribués. Le cluster des cartes Parallella offre un bon compromis entre prix et performance et peut être vu comme une seule machine à 64 coeurs.

À titre de travail futur, nous proposons d'améliorer l'architecture sur deux niveaux. Afin d'exploiter le plus de ressources, nous proposons d'ajouter un moniteur de ressources disponibles au niveau du téléphone mobile qui va s'occuper de la distribution des traitements relatifs à la détection entre la CPU et la GPU selon leur disponibilité. Dans le but de fournir un plus haut niveau de précision de détection, nous pouvons appliquer les mêmes traitements de correspondances de patrons sur d'autres données comme les permissions et les appels API et combiner les résultats obtenus.

Au niveau de l'architecture de cluster, plusieurs améliorations peuvent être établies. L'utilisation d'architectures hybrides et l'augmentation du nombre de noeuds peuvent considérablement augmenter la performance de calcul du système. L'intégration de cluster de GPUs mobiles peut également être une piste intéressante pour accélérer et alléger les traitements de sécurité sur ce type de systèmes à capacité de calcul limitée. Un moniteur pour l'architecture de cluster peut également être intégré. Ce moniteur aura pour but de distribuer les charges de données selon la disponibilité des noeuds et la charge de traitement en cours d'exécution. Enfin, nous prévoyons de porter notre plateforme sur une architecture de superordinateur puisque la solution que nous proposons est portable et extensible.

Les résultats de ce travail ont été publiés dans un article présenté en septembre 2015 au 34th IEEE Symposium on Reliable Distributed Systems Workshops (SRDSW ’15) (Voir ANNEXE I). De plus, nous prévoyons de soumettre un article journal sur la totalité du travail.

ANNEXE I

ARTICLE

Manel Abdellatif, Chamseddine Talhi, Abdelwahab Hamou-Lhadj, Michel Dagenais. 2015. On the use of mobile GPU for Accelerating Malware Detection Using Trace Analysis. In *Proceedings of IEEE 34th Symposium on Reliable Distributed Systems Workshops* (SRDSW '15). Montréal, Canada, 42-47. DOI=10.1109/SRDSW.2015.18

On the Use of Mobile GPU for Accelerating Malware Detection Using Trace Analysis

Manel Abdellatif, Chamseddine Talhi
Depart. of Software Engineering and IT
École de Technologie Supérieure
Montréal, Québec, Canada
manel.abdellatif.1@ens.etsmtl.ca
Chamseddine.Talhi@etsmtl.ca

Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University
Montréal, Québec, Canada
abdelw@ece.concordia.ca

Michel Dagenais
Department of Computer Engineering
École Polytechnique de Montréal
Montréal, Québec, Canada
michel.dagenais@polymtl.ca

Abstract—Malware detection on mobile phones involves analysing and matching large amount of data streams against a set of known malware signatures. Unfortunately, as the number of threats grows continuously, the number of malware signatures grows proportionally. This is time consuming and leads to expensive computation costs, especially for mobile devices where memory, power and computation capabilities are limited. As the security threat level is getting worse, parallel computation capabilities for mobile phones is getting better with the evolution of mobile graphical processing units (GPUs). In this paper, we discuss how we can benefit from the evolving parallel processing capabilities of mobile devices in order to accelerate malware detection on Android mobile phones. We have designed and implemented a parallel host-based anti-malware for mobile devices that exploits the computation capabilities of mobile GPUs. A series of computation and memory optimization techniques are proposed to increase the detection throughput. The results suggest that mobile graphic cards can be used effectively to accelerate malware detection for mobile phones.

Index Terms—*Malware detection, Parallel processing, Multi-pattern matching, Trace analysis of mobile phones*

I. Introduction

Smartphones or mobile phones are more and more used in personal and business life. Their popularity has made them an attractive target for malicious attacks. In fact, the number of malicious software and threats is growing significantly for mobile devices. Recently, Kaspersky Lap reported around 10 million malwares were detected between 2012 and 2013, where 98.10% of all detected malwares in 2013 targeted Android systems [1].

To ensure a high security level, a typical anti-malware matches streams of data, generated from mobile devices, against a large set of known malware signatures, using multi-matching algorithms. Most of these algorithms use a Deterministic Finite Automata (DFA) structure. The main advantage of using a DFA structure is that we can check the presence of malicious signatures in a single pass of the input data stream. As the number of malwares grows, the number of signatures also increases, hindering scalability of mobile anti-malware systems due to reduced memory and computing capabilities of mobile devices. A common solution is to offload malware detection processing to an external server. The reliance on an external server, however, exposes malware detection systems to connectivity problems.

In this paper, we explore how mobile GPUs (Graphics Processing Units), combined with parallelization techniques, can be used to design an anti-malware system that resides on the mobile device itself. Both computational capabilities and memory specifications of mobile GPUs are rapidly evolving, which makes more intensive applications and GPGPU (General-Purpose GPU) processing possible. For instance, the performance of ARM's mobile graphic cards has been improved five times in the last two years [2]. As the System-on-Chip (SoC) industry gained momentum from growing cell phone sales, ARM introduced more sophisticated GPUs like Mali T-658, which is one of the most performant ARM GPUs for mobile devices. It is scalable up to eight cores and gained up to 10x graphics performance compared to mainstream Mali-400 MP implementations [2].

In this paper, we discuss how we can effectively use the power of parallel processing on mobile GPUs to enhance the security level of the whole system. To our knowledge, this is the first time the use of mobile GPUs for security is attempted. More precisely, we show how mobile GPUs can be used to accelerate malware detection using multi-pattern matching techniques. We also investigate techniques for compacting DFA structures in order to scale over the reduced memory of mobile GPUs.

In summary, the main contributions of the paper are as follows: We have designed and implemented a high performance parallel host-based anti-malware on mobile GPU using behavioral detection techniques. We implemented a series of optimizations to deal with low memory of mobile devices and the ever-increasing computing and memory requirements of malware detection. In order to improve the performance of our prototype, we focus on the efficient use and placement of the different data in the hierarchical mobile GPU memory and reduce the average latencies of memory access.

II. Background

Multi-pattern matching algorithms have extensively been used by intrusion detection systems (IDS) and malware scanners. GPUs are used in accelerating multi-pattern matching due to their high efficiency level and the evolving computation capabilities compared to CPUs. In this section we introduce the Aho-Corasick (AC for short) [4] multi-pattern matching algorithm which is the core of our detection framework. Then, we introduce the OpenCL programming model that we used as well as some mobile malware detection paradigms.

A. Aho-Corasick Pattern Matching Algorithm

The Aho-Corasick [4] algorithm has been widely used for multiple pattern matching. This algorithm is simple and capable to locate a finite set of key words within a given input stream in a single pass. The algorithm consists of two steps: pre-processing and processing. During pre-processing, a

1060-9857/15 $31.00 © 2015 IEEE
DOI 10.1109/SRDSW.2015.18

deterministic finite automaton structure is built from given patterns. Then, in the processing step, the engine detects the presence of patterns from an input string in a single pass.

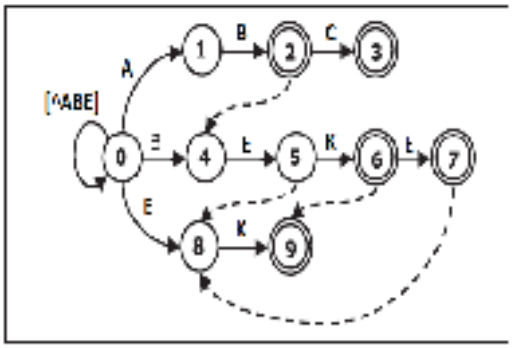


Fig. 1. AC state machine relative to patterns "AB", "ABC", "BEKE", and "EK".

In order to reduce outgoing transitions of each state, AC DFA integrates another kind of transitions, defined as failure transitions. Failure transitions are represented by dotted lines in Figure 1. They are used to backtrack any pattern that starts from any position of the input string. The DFA structure can be described by three tables: the state transition table (where valid transitions are stored), the failure transition table containing the reference of the different failure transitions in the DFA, and finally the output states table where final states are stored. Given the input character from an input stream and the current state of the finite state machine, a valid transition is checked by the machine to determine the next state. If there is no valid transition, a redirection to the state pointed by the failure transition is done. So the engine reads the same input character until it a valid transition is found.

B. *OpenCL programming model*

OpenCL [3] is an open industry standard and a framework for parallel programming of heterogeneous systems composed of devices with different capabilities such as CPUs and GPUs. The OpenCL platform model consists of a host and one or more OpenCL devices. Each device contains one or more compute unit. Each compute unit contains several processing elements. The host offloads parallel tasks to a device within special functions called kernels that are compiled at runtime by an OpenCL driver. Parallel jobs are executed by threads called work items. A hierarchical memory model is defined by OpenCL containing several types of memories like global, local and private memories. In order to achieve higher performance and maximum use of the limited computation resources on a mobile platform, partitioning the tasks between CPU and GPU, exploring efficient algorithmic parallelism, and optimizing the memory access are needed to be carefully considered.

III. ARCHITECTURE

Our framework, which is illustrated by Figure 2, is divided into three blocks: pre-processing, trace collection, and processing block.

The pre-processing block is located outside the mobile phone. It uses as input malware signatures. In our architecture, we use a combination of patterns of raw system calls reflecting malicious behaviour as signatures [5]. We can also use for example bytecode signatures or permissions' patterns. We convert malware signatures into DFA structure that will be used later by a multi-pattern matching algorithm in order to detect the presence of a malware on the device. With a DFA structure, we can check the presence of malicious signatures in a single pass of the input data stream.

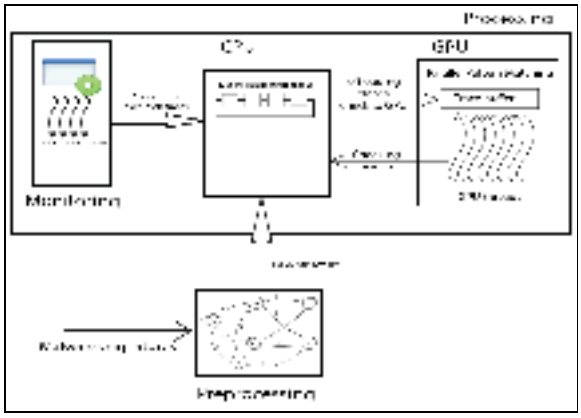


Fig. 2. The framework architecture

Figure 3 shows the transition states matrix TSM that illustrates the AC automaton structure. The row index corresponds to the automaton states and the column index denotes system calls index. Thus, for a given state *i* and an input system call *j* TSM[i][j] indicates the next state to reach. As the number of malwares grows, the number of signatures also increases. As a result, the DFA structure becomes memory consuming and needs to be compacted in order to scale with the small device memory. Many DFA compacting techniques can be exploited such as eliminating failure transitions or using special state encoding scheme to allow memory efficient use of DFA. This issue will be detailed in the next section.

The second block of our architecture is the trace collection block. It is responsible for recording application events and their behaviour by tracking the different system calls made by the processes on the embedded devices at runtime. We use for this goal the Strace tool [6], which is a useful diagnostic and debugging tool that records and intercepts system calls with their arguments. Only raw system calls are kept in our trace files. These traces will be scanned in parallel by the processing block in order to detect a malicious behaviour.

The core block of our architecture is the processing block. It is responsible for analysing application traces and communicating data to the GPU for further processing. The CPU offloads the stream of an application's traces to the GPU. The input stream is divided into several segments. Every work-group is responsible for scanning a given segment. Each byte of the input stream segment is allocated to a thread that checks the presence of malicious pattern from its starting position. If there is no valid transition, the thread terminates and starts the check from another position which is equal to the current starting position plus the total thread number of the processing block. As a result, there is no need to keep failure transitions on the DFA since GPU threads do not need to backtrack the state machine.

In this section, we discuss several optimization techniques in order to increase the performance of mobile malware detection using GPUs. These optimizations are structured into two parts: host code optimizations and kernel code optimizations.

A. *Host code optimizations*

Total memory requirements optimization: With the ever increasing number of malware signatures and the small

memory size of mobile GPUs, we need to optimize memory usage in order to achieve a high security level on a mobile phone. In our framework, three DFA structures compacting techniques are applied to deal with more malware signatures. The first one consists of eliminating failure transitions since we allocate a thread for each input string byte that checks the presence of a pattern from its start position. Then we eliminate the final states table by reordering the numbering of such states and allocating for each one a number greater than the total states number [7]. Finally, we apply P3FSM [8] algorithm on the DFA structure to have a more compacted dataset. This technique offers an effective coding of state machine and deals with the excessive memory requirement of a DFA structure.

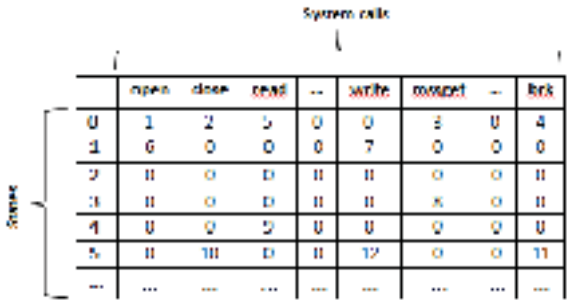


Fig. 3. State Transition Matrix

IV. Optimization Techniques

Load balancing configuration: It is an important aspect of OpenCL device processing elements to be performed on both host and device levels. At the host level, we have to choose carefully how to send trace files to the device in order to perform parallel scan. Three scenarios are considered and detailed in the next section. For the device side, every kernel is executed with the specification of the work-items distribution size. This is performed by specifying tow parameters, which are the global work size and the local work size. The first one is the number of work-items to be processed for each dimension and the second one is the number of work-items in a work-group for each dimension. To achieve high performance, it is recommended to maximize the global work size in order to fully utilize the GPU. For the local work size, the OpenCL developer's guide recommends to put it to Null if no data is shared between work-items, and let the driver device determine the most suitable work group size value. In our case, work-items are sharing data. So we have to determine explicitly local work size in order to enhance the performance of our architecture.

B. Kernel code optimizations : effective use of memory types:

Typical GPUs have hierarchical memory model architecture. There are four types of memories: global, constant, local and private. Global memory is visible to all the compute units on a device. All transfers between the host and the device are transfers to and from the device's global memory. Constant memory is also visible to all of the compute units on the device. In addition, any element of constant memory is accessible by all work-items. Local memory is a memory that belongs to a compute-unit and shared by all the work-items within a work-group. The private memory belongs to a work-item and is not accessible for other work-items. The access to constant memory is much faster than global memory access. That's why we have to choose carefully where to locate our data in order to maximize the framework performance.

V. Evaluation

In this section, we evaluate the effectiveness of our architecture to detect malwares on a mobile device using GPU and parallel processing. For our experiments, we used Qualcomm Snapdragon 801 [9], quad-core CPU at 2.5 GHz and a Qualcomm Adreno 330 GPU integrated on a Sony Xperia mobile phone. We use, as benchmark, a dataset of malicious system calls patterns that we generated from different malware families. To evaluate the performance of our framework, we measure the throughput and acceleration rate over the sequential approach. The adjustment of threads' number is studied in order to guarantee the maximum use of GPU and achieve higher throughput. To assess memory usage and throughput, several configurations are measured in our experiments. In our experiments, the system throughput is defined as:

$$Throughput = Input\ Stream\ Size / (T_{host/device}+T_{GPU}+T_{device/host}) \quad (1)$$

where $T_{host/device}$ is the transfer data time from host to the GPU, T_{GPU} is the processing time of input stream data on GPU, and $T_{device/host}$ is the transfer data time from GPU to the host.

The number of threads: In this experiment, we focus on the regulation of local work-group size. It must be specified before queuing any kernel that is executed on the device. For each configuration, we calculate the relative throughput as described by Equation (1). Figure 4 shows that the best throughput is obtained with 16 threads per work-group. Then we notice that the more we increase the number of thread, the worse the throughput.

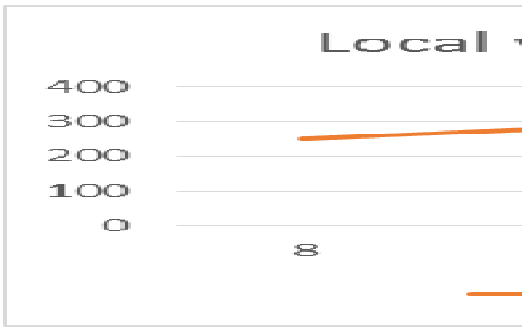


Fig. 4. Local work-group resizing

Memory types: We compared several memory configurations that are listed in Table 1. The data used are: *transition_table* which contains information about the different transitions of the DFA structure, the *input_buffer* which contains application traces that will be analyzed and the *result_buffer* that contains processing results. As results must be sent to the CPU and written to the *result_buffer,* it always must be placed on the global memory. Both constant and local memory size is much lower than the GPU global memory. As the size of *transition_table* is important, we cannot place it as a full block in the constant memory neither in local memory. Only its first part *transition_tableP1* is placed on these memory types for some configurations listed on Table 1. The second part of the table *transition_tableP2* is located in global memory for the same configurations.

Table 1. Different memory allocation configurations for the framework data structures

Memory Config.	Global Memory	Constant Memory	Local Memory
Conf1	*transition_table input_buffer result_buffer*	NA	NA
Conf2	*transition_table result_buffer*	NA	*input_buffer*
Conf3	*transition_table result_buffer*	*input_buffer*	NA
Conf4	*transition_tableP2 result_buffer*	*transition_tableP1*	*input_buffer*
Conf5	*transition_tableP2 input_buffer result_buffer*	*transition_tableP1*	NA
Conf6	*transition_tableP2 result_buffer*	*input_buffer*	*transition_tableP1*

Figure 5 shows the relative throughput of the different configurations. As we can see, the best throughput is obtained with Config4. The constant memory access time is faster than the access time of global memory. Comparing to the second configuration, with the use of constant memory, a gain of around 19% in performance is obtained. We keep this type of configuration in the following experiments.

Acceleration: We compare the performance of our framework with one that does not parallel processing. As we can see in Figure 5, all configurations of the parallel processing perform better than the serial processing. An acceleration of around three times is obtained with the parallel processing on the mobile GPU using Conf4.

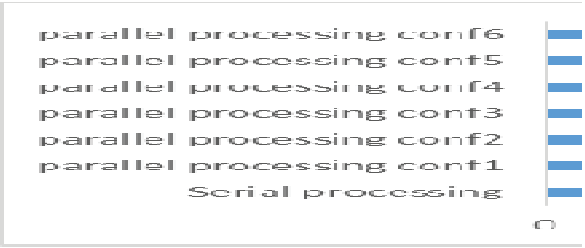


Fig. 5. Serial processing throughput vs parallel processing

Applications trace files scanning management: In this section, we experiment with different scenarios of scanning trace files for every application. The first scenario is to consider a fixed length GPU input buffer to which we send sequentially trace files. The input buffer is dedicated to only one trace file per application at the same time. Every application traces will be profiled in parallel in order to detect a malicious behavior.

The second scenario is almost the same as the first one. The only difference consists in the use of the input buffer. We send sequentially trace files to the input buffer. If, at a given time, there is a space left in the input buffer we send immediately the following application trace file. As a result the input buffer will be always fully used except at the end of scanning the final application trace file.

The third scenario is to consider also a fixed length GPU input buffer, but that has multiple entries. In other words, we send to the input buffer multiple application trace files simultaneously and profile them in parallel. In order to estimate the optimal number of applications that can be scanned in parallel we collected system calls traces from 100.

Android applications were executed normally during 3 minutes each on a Sony Xperia Z smartphone operating on Android 4.4.4. For the third scenario, we considered input stream buffers divided by 5, 10 and 15 segments, and for every buffer each segment is dedicated to an input application trace file. GPU profiling threads are equally distributed for each segment. More threads will be allocated per input buffer segment in order to have a balanced distribution of the parallel scanning processing. We found that the third scenario performs well when the input buffer is divided to 10 segments. As a result, the optimal number of applications that we can scan in parallel with this configuration is 10.

In order to study the effectiveness of the three scenarios we considered different sizes of input stream buffers. As we can see in Figure 6, the third scenario has always the best throughput. In fact, for the first scenario, there is waste of allocated resources because the input buffer is not fully exploited. Moreover for the tow first configurations more data transfers are required to process all the application traces. As a result, the processing overhead becomes more important with such scenarios. On the other hand, the parallel processing of applications trace files simultaneously gives much less processing overhead due to its effectiveness in exploiting the GPU input buffer allocation and processing. We notice that the bigger the input buffer is, the faster the total execution time become. In fact, several data large transfers between the CPU and GPU are much better than many small ones. Finally, we notice that the framework throughput is dominated by data transfers between the host and the device which consist of 70% of the total processing time.

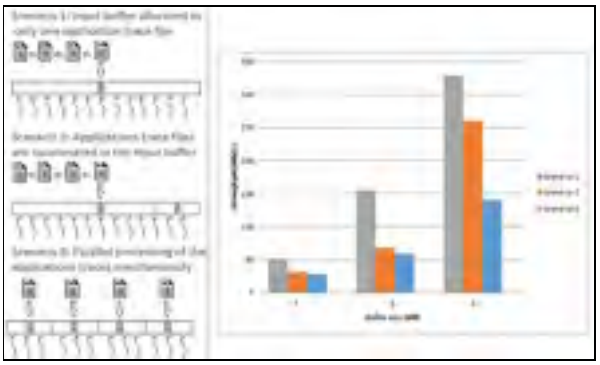


Fig. 6. Performance of the trace files scanning scenarios

Table 2. Memory requirement of PFAC and P3FSM

Number of patterns	PFAC (KB)	P3FSM (KB)
2000	67677	8922
2200	74398	9234
10000	678937	50765
16000	806554	60432
17600	809321	74380

Memory requirement: Storing a DFA structure on the GPU is memory consuming especially that mobile GPU memory is small. Table 2 lists the difference in memory requirement between PFAC DFA and P3FSM. The best result is obtained with P3FSM that compacts the DFA structure by around 10 times comparing to standard PFAC DFA. With PFAC, the limit number of patterns that we can work with on our mobile GPU is 2200 patterns. Applying P3FSM compacting technique allows as to work with 17600 patterns witch is much better.

VI. Related Work

Parallel processing techniques have been widely used over the years in order to improve the performance of multi-pattern

matching algorithms and this is due to the rapid advent of GPGPUs. Both hardware and software techniques are used to accelerate multi-pattern searching. On one hand, common hardware supported techniques use GPU [7][10][8], FPGA [11] as well as Cell/B.E processors [12]. Parallel multi-pattern matching software based approaches [13][14] exploits multicore processors to accelerate the overall processing.

Perhaps, the first implementation of parallel regular expression searching and multi-pattern matching was in Gnort [15][16]. For performance issues, Gnort utilizes a multi-pattern matching technique that uses a high memory DFA structure. A maximum throughput of 2.3 GB/s is achieved. Many studdies have been conducted to optimize the Aho-Corasick [4] string matching algorithm. An example is the Parallel Failureless-AC Algorithm (PFAC) [8]. There, all failure transitions of the DFA were removed so that there is no need to backtrack the state machine used, reducing the complexity of the algorithm. Arudchutha et al. [13] proposed an adaptation of the Aho-Corasick algorithm for multicore CPUs using POSIX thread utility. The pattern-set is divided into smaller ones and allocated for each CPU thread. This approach performs better than Herath et al. [14] implementation where they applied the same techniques but considered smaller deterministic state machines with failure transitions. Tran et al. [17] proposed a parallel implementation of the Aho-Corasick algorithm on an Nvidia desktop GPU that focuses on efficient scheduling of the off-ship global memory loads and the storage in the desktop GPU global memory. A speed-up of 222x was achieved compared to a sequential version of the same algorithm running on a 2.2Ghz Core2Duo Intel processor. Huang et al. [10] implemented a Wu-Manber-like multi-pattern matching algorithm on GPUs and achieved a maximum speed twice as fast as the modified Wu-Manber algorithm used in Snort [18]. Vasiliadis et al. [19] worked on a massively parallel antivirus engine based on ClamAV [20], called Gravity. A parallel pre-scanning of patterns is done through the Geforce GTX GPU, reaching a throughput of 40GB/s.

Many DFA compacting techniques have been proposed in the literature. Compression and bitmap [21] were proposed to improve the memory efficiency of the Aho-Corasick algorithm. Tan and Sherwood [22] introduced a memory-efficient, multi-pattern matching engine based on the bit-split techniques and string partitioning. Vespa et al [8] proposed a memory-efficient, portable and scalable string matching engine called P3FSM. One entry in memory is required for each state. The code for each state contains all the information about the possible next state.

VII. Conclusion

In this paper, we have designed and implemented a parallel host-based anti-malware for Android mobile devices based on the use of GPUs. Our framework capitalizes on the use of the highly threaded architecture of mobile GPUs, as well as the parallel nature of malware scanning to achieve end-to-end throughput in the order of 333 Mbits/s. This result is three times faster than the serial version running on a mobile CPU. To accelerate mobile malware detection on GPU, we used different types of GPU memories and explored the optimal buffer size for input streams data as well as the best scanning process scenario. In order to overcome the problem of the low memory of mobile GPUs and to deal with more malware signatures, series of memory compacting techniques were applied.

Acknowledgment

This research is partly supported by a grant from NSERC, DRDC Valcartier (QC), and Ericsson Canada.

References

[1] V. Chebyshev, and R. Unuchek, "Mobile Malware Evolution: 2013" White paper available on http://securelist.com/analysis/kaspersky-security-bulletin/58335/ mobile-malware-evolution-2013/

[2] "ARM," at http://www.arm.com/products/multimedia/mali-cost-efficient-graphics/index.php

[3] "The OpenCL 1.2 specification," at http://www.khronos.org/opencl, 2012.

[4] A. Aho and M. Corasick, 'Efficient string matching: an aid to bibliographic search', Commun. ACM, vol. 18, no. 6, pp. 333-340, 1975.

[5] Y-D Lin, Y-C Lai, C-H Chen, H-C Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," Elsevier Computers & Security, vol. 39, p. 340-350, 2013.

[6] STrace manual, Retrieved February 20, 2015, http://man7.org/linux/man-pages/man1/strace.1.html

[7] Cheng-Hung Lin; Sheng-Yu Tsai; Chen-Hsiung Liu; Shih-Chieh Chang; Shyu, J.-M., "Accelerating String Matching Using Multi-Threaded Algorithm on GPU," Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE , vol., no., pp.1,5, 6-10 Dec. 2010.

[8] Vespa, L.; Mathew, M.; Ning Weng, "P3FSM: Portable Predictive Pattern Matching Finite State Machine," Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, vol., no., pp.219, 222, 7-9 July 2009.

[9] "Qualcomm," at https://www.qualcomm.com/products/snapdragon/processors/801

[10] N. Huang, H. Hung, S. Lai, Y. Chu, and W. Tsai, "A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems," In Proc. of the International Conference on Advanced Information Networking and Applications, pp. 62-67, 2008

[11] B. W. Watson and G. Zwaan, "A taxonomy of keyword pattern matching algorithms," Computing Science Note 92/27, Eindhoven University of Technology, The Netherlands, 1992.

[12] D. P. Scarpazza, O. Villa, F. Petrini, "Exact multi-pattern string matching on the cell/be processor," In Proc. of the 5th Conference on Computing Frontiers, pp. 33-42, 2008.

[13] S. Arudchutha, T. Nishanthy, R. G. Ragel, "String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm," In Proc. of the 8th IEEE International Conference on Industrial and Information Systems, pp. 231-236, 2013.

[14] D. Herath, C. Lakmali, R. Ragel, "Accelerating string matching for bio-computing applications on multi-core CPUs," In Proc. of the 7th IEEE International Conference on Industrial and Information Systems, pp. 1-6, 2012.

[15] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," In Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection, 2008.

[16] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," In Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection, 2009.

[17] N. P. Tran, M. Lee, S. Hong, J. Choi, "High throughput parallel implementation of Aho-Corasick algorithm on a GPU," In Proc. of the IEEE 27th International Symposium on Parallel and Distributed systems (Workshops & PhD Forum), pp. 1807-1816, 2013.

[18] M. Roesch, "Snort—Lightweight Intrusion Detection for Networks," In Proc. of 15th Systems Administration Conference, 1999.

[19] G. Vasiliadis, S. Ioannidis, "Gravity: a massively parallel antivirus engine," In International Symposium on Recent Advances in Intrusion Detection, pp. 79-96, 2010.

[20] "CLAMAV," at http://www.clamav.net

[21] N. Tuck et al., "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," INFOCOM, pp. 333–40, 2004.

[22] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," In Proc. of the 32nd Annual International Symposium on Computer Architecture, pp. 112–22, 2005.

BIBLIOGRAPHIE

Aho, A. V. et M. J. Corasick. 1975. « Efficient string matching : an aid to bibliographic search ». *Communications of the ACM*, vol. 18, n° 6, p. 333–340.

Allauzen, C., M. Crochemore, et M. Raffinot. 2001. « Efficient experimental string matching by weak factor recognition* ». In *Combinatorial Pattern Matching*. p. 51–72. Springer.

Alomari, M. A. et K. Samsudin. 2011a. « A framework for GPU-accelerated AES-XTS encryption in mobile devices ». In *TENCON 2011-2011 IEEE Region 10 Conference*. p. 144–148. IEEE.

Alomari, M. A. et K. Samsudin. 2011b. « A framework for GPU-accelerated AES-XTS encryption in mobile devices ». In *TENCON 2011-2011 IEEE Region 10 Conference*. p. 144–148. IEEE.

Amamra, A., C. Talhi, et J. Robert. Oct 2012a. « Smartphone malware detection : From a survey towards taxonomy ». In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. p. 79-86.

Amamra, A., C. Talhi, et J. Robert. 2012b. « Performance Evaluation of Multi-pattern Matching Algorithms on Smartphone ». In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2012 Seventh International Conference on*. p. 329–334. IEEE.

Arica, N. et F. T. Yarman-Vural. 2002. « Optical Character Recognition for Cursive Handwriting ». *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, n° 6, p. 801-813.

ARM. 2015. « ARM GPUs ». <http://www.arm.com/products/multimedia/mali-gpu/high-area-efficiency/mali-t820-t830.php>.

Arudchutha, S., T. Nishanthy, et R. G. Ragel. 2013. « String matching with multicore CPUs : Performing better with the Aho-Corasick algorithm ». In *Industrial and Information Systems (ICIIS), 2013 8th IEEE International Conference on*. p. 231–236. IEEE.

Aysan, A. I. et S. Sen. 2015. « API call and permission based mobile malware detection ». In *Signal Processing and Communications Applications Conference (SIU), 2015 23th*. p. 2400–2403. IEEE.

Baktir, S. et E. Savaş. 2013. Highly-parallel montgomery multiplication for multi-core general-purpose microprocessors. *Computer and Information Sciences III*, p. 467–476. Springer.

Barak, A. et A. Shiloh. 1999. « The MOSIX Cluster Management System for Distributed Computing on Linux Clusters and Multi-Cluster Private Clouds ».

Bellekens, X., I. Andonovic, R. Atkinson, C. Renfrew, et T. Kirkham. 2013. « Investigation of GPU-based pattern matching ». In *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*.

Bora, P. et T. Czajka. 1999. « Implementation of the SERPENT Algorithm using ALTERA FPGA Devices ». *Public Comments on AES Candidate Algorithms-Round*, vol. 2.

Bourges-Sevenier, M. 2013. « Aptina ». <http://www.slideshare.net/mikeseven/imaging-on-embedded-gp-us-bamm-meetup-20131219>.

Chan, P. P. et W.-K. Song. 2014. « Static detection of Android malware by using permissions and API calls ». In *Machine Learning and Cybernetics (ICMLC), 2014 International Conference on*. p. 82–87. IEEE.

Cheng-Hung Lin,Sheng-Yu Tsai, C.-H. L. S.-C. C. et J.-M. Shyu. 2013. « Accelerating String Matching Using Multi-threaded Algorithm on GPU ». *IEEE Transactions*, vol. 62, n° 10, p. 1906-1916.

Christodorescu, M. et S. Jha. 2006. *Static analysis of executables to detect malicious patterns*. Technical report.

Commentz-Walter, B., 1979. *A string matching algorithm fast on the average*.

Dai, S., Y. Liu, T. Wang, T. Wei, et W. Zou. 2010. « Behavior-based malware detection on mobile phone ». In *Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference on*. p. 1–4. IEEE.

Damaj, I. W. 2007. « Parallel algorithms development for programmable devices with application from cryptography ». *International Journal of Parallel Programming*, vol. 35, n° 6, p. 529–572.

Davis, S., B. Jones, et H. Jiang. 2015. « Portable parallelized blowfish via RenderScript ». In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*. p. 1–6. IEEE.

Dongara, P. et T. Vijaykumar. 2003. « Accelerating private-key cryptography via multithreading on symmetric multiprocessors ». In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*. p. 58–69. IEEE.

Fan, W., Y. Liu, et B. Tang. 2015. « An API Calls Monitoring-based Method for Effectively Detecting Malicious Repackaged Applications ». *International Journal of Security and Its Applications*, vol. 9, n° 8, p. 221–230.

Flynn, M. J. 1972. « Some computer organizations and their effectiveness ». *Computers, IEEE Transactions on*, vol. 100, n° 9, p. 948–960.

Gascon, H., F. Yamaguchi, D. Arp, et K. Rieck. 2013. « Structural detection of android malware using embedded call graphs ». In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. p. 45–54. ACM.

Giorgi, P., L. Imbert, et T. Izard. 2013. « Parallel modular multiplication on multi-core processors ». In *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*. p. 135–142. IEEE.

Guo, W., Y. Liu, S. Bai, J. Wei, et D. Sun. 2012. « Hardware architecture for RSA cryptography based on residue number system ». *Transactions of Tianjin University*, vol. 18, p. 237–242.

Herath, D., C. Lakmali, et R. Ragel. 2012. « Accelerating string matching for bio-computing applications on multi-core CPUs ». In *Industrial and Information Systems (ICIIS), 2012 7th IEEE International Conference on*. p. 1–6. IEEE.

Howard, R. 1987. « Data encryption standard ». *Information age*, vol. 9, n° 4, p. 204–210.

Huang, N.-F., H.-W. Hung, S.-H. Lai, Y.-M. Chu, et W.-Y. Tsai. 2008. « A gpu-based multiple-pattern matching algorithm for network intrusion detection systems ». In *Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on*. p. 62–67. IEEE.

Isohara, T., K. Takemori, et A. Kubota. 2011. « Kernel-based behavior analysis for android malware detection ». In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*. p. 1011–1015. IEEE.

Kaewkasi, C. et W. Srisuruk. 2014a. « Optimizing performance and power consumption for an ARM-based big data cluster ». In *TENCON 2014-2014 IEEE Region 10 Conference*. p. 1–6. IEEE.

Kaewkasi, C. et W. Srisuruk. 2014b. « A study of big data processing constraints on a low-power Hadoop cluster ». In *Computer Science and Engineering Conference (ICSEC), 2014 International*. p. 267–272. IEEE.

Kaihara, M. E. et N. Takag. 2008. « Bipartite modular multiplication method ». *Computers, IEEE Transactions on*, vol. 57, n° 2, p. 157–164.

Kouzinopoulos, C. S., P. D. Michailidis, et K. G. Margaritis. 2012. « Performance Study of Parallel Hybrid Multiple Pattern Matching Algorithms for Biological Sequences. ». In *BIOINFORMATICS*. p. 182–187.

Lee, S.-W. et J.-L. Gaudiot. 2003. Clustered microarchitecture simultaneous multithreading. *Euro-Par 2003 Parallel Processing*, p. 576–585. Springer.

Lin, C.-H., C.-H. Liu, S.-C. Chang, et W.-K. Hon. 2012. « Memory-efficient pattern matching architectures using perfect hashing on graphic processing units ». In *INFOCOM, 2012 Proceedings IEEE*. p. 1978–1986. IEEE.

Lin, K.-J., Y.-H. Huang, et C.-Y. Lin. 2013a. Efficient parallel knuth-morris-pratt algorithm for multi-gpus with cuda. *Advances in Intelligent Systems and Applications-Volume 2*, p. 543–552. Springer.

Lin, Y.-D., Y.-C. Lai, C.-H. Chen, et H.-C. Tsai. 2013b. « Identifying android malicious repackaged applications by thread-grained system call sequences ». *computers & security*, vol. 39, p. 340–350.

Lin, Z. et P. Chow. 2013. « Zcluster : A zynq-based hadoop cluster ». In *Field-Programmable Technology (FPT), 2013 International Conference on*. p. 450–453. IEEE.

Loghin, D., B. M. Tudor, H. Zhang, B. C. Ooi, et Y. M. Teo. 2015. « A performance study of big data on small nodes ». *Proceedings of the VLDB Endowment*, vol. 8, n° 7, p. 762–773.

Mandumula, K. K. 2011. « Knuth-Morris-Pratt Algorithm ». *Poslední zmena*, vol. 18.

Martins, P. et L. Sousa. 2014. « On the evaluation of multi-core systems with simd engines for public-key cryptography ». In *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*. p. 48–53. IEEE.

Miller, F. P., A. F. Vandome, et J. McBrewster. 2009. « Advanced Encryption Standard ».

MuthuKumar, B. et S. Jeevananthan. 2010. « High speed hardware implementation of an elliptic curve cryptography (ECC) co-processor ». In *Trendz in Information Sciences & Computing (TISC), 2010*. p. 176–180. IEEE.

Nagendra, M. et M. C. Sekhar. 2014. « Performance Improvement of Advanced Encryption Algorithm using Parallel Computation ». *International Journal of Software Engineering and Its Applications*, vol. 8, n° 2, p. 287–296.

Nvidia. 2015. « Nvidia Tegra X1 ». <http://www.nvidia.com/object/tegra-x1-processor.html>.

OpenCL. 2015. « OpenCL 2.1 ». <https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf>.

Ou, Z., B. Pang, Y. Deng, J. K. Nurminen, A. Yla-Jaaski, et P. Hui. 2012. « Energy-and cost-efficiency analysis of arm-based clusters ». In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. p. 115–123. IEEE.

Padua, D., 2011. *Encyclopedia of parallel computing*, volume 4.

Peiravian, N. et X. Zhu. 2013. « Machine learning for android malware detection using permission and api calls ». In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*. p. 300–305. IEEE.

Ping, X., W. Xiaofeng, N. Wenjia, Z. Tianqing, et L. Gang. 2014. « Android malware detection with contrasting permission patterns ». *Communications, China*, vol. 11, n° 8, p. 1–14.

Pungila, C. 2013. « Hybrid Compression of the Aho-Corasick Automaton for Static Analysis in Intrusion Detection Systems ». In *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions*. p. 77–86. Springer.

Pungila, C. et V. Negru. 2012. A highly-efficient memory-compression approach for gpu-accelerated virus signature matching. *Information Security*, p. 354–369. Springer.

Qin, Z., Y. Xu, B. Liang, Q. Zhang, et J. Huang. 2013. « An Android malware static detection method ». *Journal of Southeast University (Natural Science Edition)*, vol. 43, p. 7–1162.

Qualcomm. 2015. « qualcomm GPUs ». <https://www.qualcomm.com/products/snapdragon/gpu>.

Rastogi, V., Y. Chen, et X. Jiang. 2013. « Droidchameleon : evaluating android anti-malware against transformation attacks ». In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. p. 329–334. ACM.

Scarpazza, D. P., O. Villa, et F. Petrini. 2008. « Exact multi-pattern string matching on the cell/be processor ». In *Proceedings of the 5th conference on Computing frontiers*. p. 33–42. ACM.

Schatz, M. et C. Trapnell. 2007. « Fast exact string matching on the GPU ». *Center for Bioinformatics and Computational Biology*.

Schmidt, A.-D., R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. Yüksel, S. Camtepe, S. Albayrak, et al. 2009. « Static analysis of executables for collaborative malware detection on android ». In *Communications, 2009. ICC'09. IEEE International Conference on*. p. 1–5. IEEE.

Sharma, A. et S. K. Dash. 2014. Mining api calls and permissions for android malware detection. *Cryptology and Network Security*, p. 191–205. Springer.

Sharma, R. et P. Kanungo. 2014. « Dynamic load balancing algorithm for heterogeneous multi-core processors cluster ». In *Communication Systems and Network Technologies (CSNT), 2014 Fourth International Conference on*. p. 288–292. IEEE.

Talha, K. A., D. I. Alper, et C. Aydin. 2015. « APK Auditor : Permission-based Android malware detection system ». *Digital Investigation*, vol. 13, p. 1–14.

Tinetti, F. G. 2000. « Parallel programming : techniques and applications using networked workstations and parallel computers ». *Journal of Computer Science & Technology*.

Tran, N.-P., M. Lee, S. Hong, et D. H. Choi. 2014. Multi-stream parallel string matching on kepler architecture. *Mobile, Ubiquitous, and Intelligent Computing*, p. 307–313. Springer.

Tumeo, A. et O. Villa. 2010. « Accelerating DNA analysis applications on GPU clusters ». In *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*. p. 71–76. IEEE.

Tumeo, A., O. Villa, et D. G. Chavarría-Miranda. 2012. « Aho-Corasick string matching on shared and distributed-memory parallel architectures ». *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, n° 3, p. 436–443.

Van Lunteren, J. et al. 2006. « High-Performance Pattern-Matching for Intrusion Detection. ». In *Infocom*. p. 1–13. Citeseer.

Vanderlei de Arruda, T., Y. R. Venturini, et T. C. Sakata. 2015. « Performance evaluation of ECC scalar multiplication using parallel modular algorithms on mobile devices ». In *Privacy, Security and Trust (PST), 2015 13th Annual Conference on*. p. 153–156. IEEE.

Vasiliadis, G. et S. Ioannidis. 2010. « Gravity : a massively parallel antivirus engine ». In *Recent Advances in Intrusion Detection*. p. 79–96. Springer.

Vasiliadis, G., M. Polychronakis, S. Antonatos, E. P. Markatos, et S. Ioannidis. 2009. « Regular expression matching on graphics hardware for intrusion detection ». In *Recent Advances in Intrusion Detection*. p. 265–283. Springer.

Vasiliadis, G., M. Polychronakis, et S. Ioannidis. 2011. « MIDeA : a multi-parallel intrusion detection architecture ». In *Proceedings of the 18th ACM conference on Computer and communications security*. p. 297–308. ACM.

Venkatachary, S. 21 2009. « Memory optimized pattern searching ». US Patent 7,565,380.

Vespa, L. et N. Weng. 2012. « SWM : Simplified Wu-Manber for GPU-based Deep Packet Inspection ». In *SAM'12 : Proceedings of The 2012 International Conference on Security and Management*. p. 150–155. Citeseer.

Vespa, L., M. Mathew, et N. Weng. 2009. « P3fsm : Portable predictive pattern matching finite state machine ». In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. p. 219–222. IEEE.

Vespa, L. J. et N. Weng. 2011. « Gpep : Graphics processing enhanced pattern-matching for high-performance deep packet inspection ». In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*. p. 74–81. IEEE.

Wang, Z. et F. Wu. 2015. « Android malware analytic method based on improved multi-level signature matching ». In *Information Science and Technology (ICIST), 2015 5th International Conference on*. p. 93–98. IEEE.

Wu, D.-J., C.-H. Mao, T.-E. Wei, H.-M. Lee, et K.-P. Wu. 2012. « Droidmat : Android malware detection through manifest and api calls tracing ». In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. p. 62–69. IEEE.

Wu, S., U. Manber, et al. 1994. « A fast algorithm for multi-pattern searching ».

Yang, Y., Z. Guan, H. Sun, et Z. Chen. 2015. Accelerating rsa with fine-grained parallelism using gpu. *Information Security Practice and Experience*, p. 454–468. Springer.

Yang, Y.-H. E. et V. K. Prasanna. 2013. « Robust and Scalable String Pattern Matching for Deep Packet Inspection on Multicore Processors ». *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, n° 11, p. 2283–2292.

Zeng, H., Y. Ren, Q.-X. Wang, N.-Q. He, et X.-Y. Ding. 2014. « Detecting malware and evaluating risk of app using Android permission-API system ». In *Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2014 11th International Computer Conference on*. p. 440–443. IEEE.

Zha, X. et S. Sahni. 2011. « Multipattern string matching on a GPU ». In *Computers and Communications (ISCC), 2011 IEEE Symposium on*. p. 277–282. IEEE.

Zhang, H., D.-f. Zhang, et X.-a. Bi. 2012. « Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU ». *International Journal of Information and Education Technology*, vol. 2, n° 2, p. 185.

Zou, S., J. Zhang, et X. Lin. 2015. « An effective behavior-based Android malware detection system ». *Security and Communication Networks*, vol. 8, n° 12, p. 2079–2089.

ÉTS. 2010. « Site web de l'ÉTS ». <http://www.etsmtl.ca>.