UNIVERSITÉ DE MONTRÉAL

APPLICATION DE L'ALGORITHME DE MAX-HASHING POUR LE RÉFÉRENCEMENT DE FICHIERS VIDÉO ET LA DÉTECTION DE CONTENUS ET DE FLUX CONNUS À HAUTE VITESSE SUR GPU

ADRIEN LARBANET
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

APPLICATION DE L'ALGORITHME DE MAX-HASHING POUR LE RÉFÉRENCEMENT DE FICHIERS VIDÉO ET LA DÉTECTION DE CONTENUS ET DE FLUX CONNUS À HAUTE VITESSE SUR GPU

présenté par : LARBANET Adrien
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. LANGLOIS Pierre, Ph.D., président
M. DAVID Jean Pierre, Ph.D., membre et directeur de recherche
M. SAVARIA Yvon, Ph.D., membre

DÉDICACE

« Quand on remporte le tour, à Sloubi, on a quatorze solutions possibles : Soit on annule le tour, soit on passe, soit on change de sens, soit on recalcule les points, soit on compte, soit on divise par six, soit on jette les bouts de bois de quinze pouces – ça, c'est quand on joue avec les bouts de bois –, soit on se couche, soit on joue sans atouts, et après, il y a les appels : plus un, plus deux, attrape l'oiseau, rigodon ou "chante Sloubi". Nous, on va faire que "chante Sloubi". »

Perceval dans *Kaamelott*, Livre III épisode 17

REMERCIEMENTS

Je voudrais tout d'abord remercier mon directeur de recherche, le professeur Jean Pierre David, pour l'encadrement de ma maîtrise. Ces trois années passées sous sa supervision furent les plus intéressantes et formatrices de mes études. J'ai toujours pu profiter de sa disponibilité, de ses conseils avisés, et surtout de son écoute aux problèmes et aux interrogations que j'ai pu rencontrer. Enfin, son implication dans le suivi de mes travaux a créé un cadre stimulant pour la réalisation du présent document.

Mes remerciements vont aussi au Ministère des Finances et de l'Économie du Québec. Les financements attribués au projet m'ont permis de travailler avec des équipements de grande qualité dont j'ai tenté d'extraire les meilleurs résultats possibles. Quant aux financements dont j'ai pu profiter directement, j'ai eu grâce à eux la possibilité me dédier intégralement à ma recherche. Je tiens aussi à adresser mes remerciements à Netclean et notamment à Daniel Jabler, Mattias Shamlo et Björn Samvik. La qualité de leur supervision n'a eu d'égal que leur gentillesse. Je suis très heureux d'avoir pu participer à leur projet et de les avoir aidés dans la cause qu'ils défendent.

J'aimerais remercier tous les étudiants que j'ai pu rencontrer sous la direction du professeur David : Tarek Ould Bachir, Matthieu Courbariaux, Marc-André Daigneault, Himan Khandazi et Jonas Lerebours dont le côtoiement m'a considérablement aidé dans l'avancement de mes travaux ainsi que dans mon développement personnel. Ils ont contribué à former une ambiance de travail amicale et chaleureuse.

Merci à mes parents Anne-Valérie et Patrice Larbanet, à mes sœurs Charlotte et Stéphanie et à mes nièces Chloé et Léa. Plus généralement, merci à ma famille entière pour son soutien solide malgré la distance qui nous a séparés pendant ces trois années. Je voudrais aussi remercier mes amis de France et du Québec notamment Jérôme Gaveau, Simon Guigui, Sophie Lallemant, Alexandre Leuliet, Tiffany Prin et Aurélien Salomon. J'adresse enfin mes remerciements particuliers à Sylvain Terlutte et Naysan Saran.

RÉSUMÉ

Le nombre croissant d'utilisateurs d'Internet et le développement de la technologie des communications s'accompagnent par l'émergence de comportements illégaux sur le réseau. Notons par exemple le partage de fichiers protégés par le droit d'auteur, comme des films ou de la musique, mais aussi le trafic d'images et de films à contenu pédo-pornographique illégal lui aussi. Il se forme alors le besoin de pouvoir efficacement cibler et détecter le transfert de ces fichiers connus sur un lien réseau.

Le fonctionnement des communications sur le réseau Internet présente de multiples problématiques. Les fichiers sont segmentés en paquets et ceux-ci empruntent des routes possiblement indépendantes pour atteindre la même destination. D'autre part, ces paquets suspicieux peuvent être noyés dans les flux de données (légaux) de millions d'autres utilisateurs rendant donc la reconstitution du fichier original depuis les paquets observés très improbable. Dans ce cas, on peut considérer que deux paquets successifs sont indépendants. Nous sommes donc contraints de travailler au niveau des paquets et, pris individuellement les uns des autres, il nous faudrait déterminer avec le maximum de certitude si ceux-ci contiennent un segment de fichier connu ou si ils appartiennent à un flux de paquets connu. Enfin, les grands réseaux utilisant des liens à 10, 40 voire 100 Gb/s, ces détections doivent pouvoir être appliquées sur des millions de paquets par seconde.

On peut simplement déterminer le flux auquel appartient un paquet en se basant sur les données de son entête. Le processus se complique lorsque l'on cherche à analyser le contenu des paquets. On est alors face à une grande quantité de données, très segmentée, dans laquelle on cherche des correspondances à des schémas (ou règles) connu(e)s. Dans ce domaine particulier, les expressions régulières sont très utilisées notamment grâce à leur versatilité. On note en revanche que les performances de celles-ci se dégradent avec le nombre de règles qu'on leur soumet. Les fonctions de hachage n'ont généralement pas ce désavantage puisque le temps d'accès à un élément d'une table est constant quelque soit l'état de remplissage de cette dernière. Elles ont, d'autre part, l'avantage de générer des empreintes de taille fixe à partir desquelles on ne peut pas reconstituer le fichier original (injectivité). Nous avons choisis de baser notre système sur l'algorithme de *max-Hashing* du fait de sa capacité à pouvoir détecter des segments de fichiers. Cette caractéristique n'est, en effet, pas partagée par tous les algorithmes de hachage et est cruciale dans le contexte que nous visons : la détection de fragments de fichiers connus, préalablement référencés, contenus dans des paquets réseau.

Le référencement de fichiers informatiques avec l'algorithme de *max-Hashing* natif, ou "brut", a tendance à générer des empreintes peu originales : celles-ci peuvent être retrou-

vées lors du référencement d'autres fichiers (différents) ou, pire, lors de l'analyse de paquets transportant des données inconnues. On parlera de redondances lors du référencement et de faux-positifs lors de la détection. Ces cas de figure sont particulièrement courants avec des fichiers de même format. Le formatage introduit en général des champs de données qui peuvent se retrouver très similairement voire identiquement d'un fichier à l'autre. Par extension, certains champs de données sont plus originaux : il sera très peu probable de les retrouver dans d'autres fichiers. Ces segments nous donnant beaucoup plus d'information sur l'identité du fichier dont ils ont été extraits, on parle de segments à "haute-entropie".

On propose donc une méthode d'extraction de ces zones à "haute-entropie" dans les fichiers vidéo utilisant le format MP4 et l'encodeur H.264 (les plus utilisés aujourd'hui) dans le but de générer des empreintes originales. Avec l'utilisation d'une base de données de 8073 fichiers vidéo, nous montrons que notre méthode permet de nullifier la probabilité de générer des empreintes redondantes, lors du référencement, ou de voir apparaitre un faux-positif lors de la détection.

Les tâches de détection des fichiers et des flux connus sont réalisées sur un GPU, situé sur une carte connectée au bus PCIe 2.0 du système. Ce genre de plateforme est particulièrement optimisé pour le traitement de grandes quantités de données en parallèle, comme nos paquets par exemple. Les contraintes principales lors du codage d'un programme pour le GPU se trouvent dans son architecture. Il faut en effet prendre en considération tous les niveaux de mémoire qui sont disponibles sur la carte ainsi que la manière dont est exécuté le code dans le GPU. Tous les fils d'exécution (ou *threads* en anglais) doivent, par exemple, suivre les mêmes instructions pour pouvoir être exécutés en parallèles. En cas de divergences dans le code, les performances du GPU chutent considérablement et les bénéfices d'une telle plateforme disparaissent.

Notre système de détection est constitué de plusieurs fonctions ou noyaux (*kernels*), terme spécifique au GPU. Dans le premier d'entre eux, on réalise l'analyse des entêtes des paquets en parallèle. On en extrait les informations importantes telles que les adresses IP, les ports et les positions des champs de données. Le second noyau hache les données selon l'algorithme de *max-Hashing* et génère des empreintes depuis celles-ci. Enfin, lors du troisième et du quatrième noyau respectivement, on compare les empreintes calculées et les informations de l'entête aux références enregistrées dans la base de données. Notre implémentation de ces noyaux permet de traiter près de 52 Gb/s, malheureusement le bus PCIe 2.0, par lequel on achemine les paquets à la mémoire du GPU, limite le système complet à 50 Gb/s.

ABSTRACT

The increasing number of Internet users and the development in communication technology are followed by the emergence of illegal behaviours on the network. For instance, we can mention the sharing of files that are protected by copyrights, such as films or music, and additionally the traffic of illegal images and video-clips related to child-pornography. There is, therefore, a need for a system that can efficiently detect the transfer of known illegal files over the Internet.

Such a system is not trivial: the protocols used on the networks imply some difficulties. Files are fragmented into packets and these may follow different path through the network to eventually reach the same destination. Moreover, the suspicious packets may get mixed with legit data streams coming from millions of other users. Hence, the reconstruction of the original file from the packets we can capture from a node is very unlikely. In this case, we have to consider that two successive packets are independent. Focusing on each packet individually and yet with the maximum certainty, we must be able to determine whether or not this packet contains parts of a known file or belongs to a known packet flow. Finally, as the technology used in large networks, such as Internet Service Providers (ISP) or Internet backbones, reaches 10, 40 and 100 Gbps bandwidths, the ability for our system to treat such bandwidth is mandatory.

One can simply determine the flow to which a packet belongs from the data embedded in its header. The process usually becomes much more complicated when trying to analyze the content of a packet. Then, we face large amount of data, very segmented, in which we try to locate the parts that match given patterns (or rules). Regular expressions are widely used for such problems thanks to their versatility but, on the other hand, their performances are bound to the number of patterns (rules) they try to locate: the more you add rules the less you can expect good performance from the system. Hashing methods usually do not have this disadvantage since the time to access an item in a hash-table is constant regardless of the its filling. Furthermore, they have the advantage of generating fixed size footprints from which we cannot reconstruct the original file (due to their injectivity property). We chose to base our system on the Max-Hashing algorithm due to its ability to detect segments of known files. This feature is not common to all hashing algorithms and is crucial for the purpose we seek: the detection of fragments of known files, previously referenced, contained in the payload of network packets.

Basically referencing computer files with the max-Hashing algorithm leads to the generation of non-original fingerprints. These may be generated again while referencing other

files (we call them redundancies) or, worse, while analyzing packet carrying unknown ones (we will call them false-positives). This kind of event can become particularly common when dealing with files that have the same format. Indeed, formatting introduces fields of data and flags that can be found in various files with little or no difference at all. By extension, some fields are more original : the kind of segment of data that is very unlikely to find in any other file. These segments give many information on the identity of the file from which they were extracted. We therefore call them "high-entropy" segments.

Thus, we propose a method for the extraction of "high-entropy" segments in video files using the MP4 format and the H.264 encoder (the most used). Once located, we can focus on these zones in order to generate original fingerprints. Using a database counting 8073 video files, we measure that our method can nullify, both, the probability of generating redundant fingerprints while referencing, and the false-positive rate while detecting.

The detection tasks for known files and known flows are performed with a GPU. The chip is deported on a card and connected to the rest of the system by a PCIe 2.0 bus. This kind of platform is particularly optimized for the processing of large amounts of data in parallel, like our network packets. But the main constraints in coding a program for the GPU lie in its architecture: there are several levels of memories on the card itself and the instructions are executed in a peculiar way. For instance, all threads must follow the same execution path in order for them to be executed in parallel. In case of divergences in the code, the GPU performances drop considerably and the benefits of such a platform disappear.

Our detection system consists of several functions or "kernels", as we call functions that are designed to be executed on a GPU. In the first kernel, the packet headers are parsed in parallel. Information such as IP addresses, TCP/UDP ports and payload offset and length are extracted and stored for further use. The second kernel hashes the payload according to the max-Hashing algorithm and generates fingerprints from them. Finally, in the third and fourth kernels, the computed fingerprints and the TCP/UDP-IP data are compared with the reference database. Our implementation of these four kernels can treat up to 52 Gbps of network packets. Unfortunately the PCIe 2.0 bus, which forwards the packets to the GPU memory, caps the system to 50 Gbps.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xii

LISTE DES FIGURES xiii

LISTE DES SIGLES ET ABRÉVIATIONS xiv

CHAPITRE 1 INTRODUCTION 1
1.1 De l'évolution des réseaux 1
1.2 Analyse des échanges sur le réseau : raisons et méthodes 2
1.2.1 Exemples de pratiques et de comportements illicites sur Internet 2
1.2.2 Détection de contenus potentiellement illégaux 3
1.3 Cahier des charges de notre système et objectifs de recherche 4
1.4 Plan du mémoire 5

CHAPITRE 2 REVUE DE LITTÉRATURE 6
2.1 Les protocoles des réseaux informatiques 6
2.1.1 Couche Liaison : Ethernet 6
2.1.2 Couche Réseau : le protocole IP 8
2.1.3 Couche Transport : protocoles TCP et UDP 8
2.2 Systèmes de détection d'intrusion 10
2.2.1 Principe 10
2.2.2 Détection d'anomalies 11
2.2.3 Détection d'abus 11
2.3 Détection de données connues 11
2.3.1 Comparaison de chaînes de caractères 12

2.3.2 Hacher pour représenter les données 16
2.3.3 Max-Hashing 19
2.4 Matériels utilisés dans le domaine d'analyse des flux réseaux 20
2.5 Algorithmes de référencement et de détection dédiés aux fichiers vidéo 21
2.6 Conclusion du chapitre 22

CHAPITRE 3 RÉFÉRENCEMENT DE FICHIERS VIDÉO 23
3.1 Originalité des segments de données dans un fichier 23
3.1.1 Fichiers textuels, analogie avec le langage 23
3.1.2 Fichiers HTML, le cas des balises 24
3.1.3 Formats de fichiers et encapsulation 25
3.2 Fichiers vidéo : étude du format MP4 avec H.264 25
3.2.1 Conteneur vidéo MP4 26
3.2.2 Codage et compression vidéo avec H.264 27
3.3 Utilisation du logiciel `FFmpeg` pour le calcul des signatures 28
3.3.1 Modification des fichiers sources de `FFmpeg` 29
3.3.2 Calcul des signatures 31
3.4 Méthodologie pour la mesure des redondances et des faux-positifs 31
3.4.1 Méthode de référencement "brute" 32
3.4.2 Méthode améliorée du référencement 33
3.5 Discussion 33
3.6 Conclusion du chapitre 33

CHAPITRE 4 SYSTÈME DE DÉTECTION 35
4.1 Réception et copies des paquets Ethernet 35
4.1.1 Interface réseau et CPU 35
4.1.2 Agrégation des paquets puis envoi vers le GPU 36
4.2 Processeurs graphiques (GPU) et programmation 37
4.2.1 Architecture du processeur graphique Kepler GK110 38
4.2.2 Programmation logicielle avec CUDA C/C++ 40
4.3 Noyaux implémentés 43
4.3.1 Analyse des entêtes Ethernet, IP et TCP/UDP 43
4.3.2 Hachage des données des paquets 44
4.3.3 Comparaison des empreintes avec la base de données 46
4.3.4 Reconnaissance des flux TCP/UDP-IP 46
4.3.5 Synthèse 47
4.4 Résultats pour les transferts sur le PCIe et les noyaux du GPU 48

4.4.1 Copie vers et depuis la mémoire embarquée du GPU 48
4.4.2 Analyse des entêtes 49
4.4.3 Recherches dans les bases de données 50
4.4.4 Hachage des champs de données 52
4.5 Système complet et *pipeline* 54
4.5.1 Conclusion du chapitre 57

CHAPITRE 5 CONCLUSION 58
5.1 Synthèse des travaux 58
5.2 Limitation de la solution proposée 59
5.3 Améliorations futures 59
5.3.1 Capture des paquets 59
5.3.2 Extension du référencement dédié 60
5.3.3 Bus PCIe 3.0 60
5.3.4 Autres GPU 60
5.3.5 Solution à multiple GPU 61

RÉFÉRENCES 62

LISTE DES TABLEAUX

Tableau 2.1 Modèle OSI 6
Tableau 2.2 Exemples de valeurs Ethertype et protocoles correspondants 8
Tableau 2.3 Complexité des algorithmes de comparaison de chaînes de caractères . . 15
Tableau 3.1 Résultats des deux méthodes de référencement 32
Tableau 4.1 Comparaison des performances (précision simple) entre CPU et GPU . 38

LISTE DES FIGURES

Figure 2.1 Trame Ethernet 7
Figure 2.2 Trame IPv4 9
Figure 2.3 Trame TCP 10
Figure 2.4 Trame UDP 10
Figure 2.5 Approche naïve pour la comparaison de chaînes de caractères 12
Figure 2.6 Application de l'algorithme de Knuth-Pratt-Morris 14
Figure 2.7 Application de l'algorithme de Rabin-Karp 14
Figure 2.8 Illustration d'un arbre de Merkle sur 4 segments 18
Figure 2.9 Illustration de l'algorithme de *winnowing* 19
Figure 2.10 Illustration de l'algorithme de *max-Hashing* 20
Figure 3.1 Exemple de code HTML5 24
Figure 3.2 Structure d'un fichier MP4 élémentaire 26
Figure 3.3 Encapsulation des données avec H.264 28
Figure 3.4 Prototypes des fonctions de `FFmpeg` (`Libav`) utilisées 30
Figure 3.5 Ressources ajoutées à la bibliothèque de `FFmpeg` 30
Figure 4.1 Copies des paquets au travers du système 35
Figure 4.2 Prototype de la fonction d'allocation de mémoire de CUDA 37
Figure 4.3 Mémoires GPU 39
Figure 4.4 Fonction en C/C++ et noyau équivalent en CUDA C/C++ 41
Figure 4.5 Organisation des fils d'exécution, des blocs et de la grille 42
Figure 4.6 Hachage d'un paquet par parts avec recouvrement 46
Figure 4.7 Mesure du temps avec les *events* CUDA 48
Figure 4.8 Bande passante observée en fonction de la taille des données (H→D) . 49
Figure 4.9 Analyse des entêtes 50
Figure 4.10 Recherche des empreintes et des flux dans les tables 51
Figure 4.11 Filtre sommaire de caractères (octets) 52
Figure 4.12 Filtre amélioré de caractères (octets) 53
Figure 4.13 Performances du hacheur dans 3 configurations différentes 54
Figure 4.14 Enchainement des noyaux 55
Figure 4.15 Performances de toute la partie de traitement (noyaux) 55
Figure 4.16 Copie d'écran du profileur NVIDIA (1) : étages du *pipeline* 56
Figure 4.17 Copie d'écran du profileur NVIDIA (2) : occupations du bus et du GPU 56

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application-specific integrated circuit
CPU	Central Processing Unit
CRC	Contrôle de Redondance Cyclique
CTPH	Context-Triggered Piecewise Hashing
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DPI	Deep Packet Inspection
DRAM	Dynamic Random Access Memory
FAI	Fournisseur(s) d'accès à Internet
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
GPU	Graphics Processing Unit
GPGPU	General-purpose Processing on Graphics Processing Unit
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPS	Intrusion Prevention System
JPEG	Joint Photographic Experts Group
LAN	Local Area Network
MAC	Media Access Control
MD5	Message Digest 5
MP	Multi-Processor
MPEG	Moving Picture Experts Group
MTU	Maximum Transmission Unit
OSI	Open Systems Interconnection
PCI(e)	Peripheral Component Interconnect (Express)
PDF	Portable Document Format
SCP	Secure Copy Protocol
SHA-1	Secure Hash Algorithm-1

SIMD	Single Instruction Multiple Data
SPI	Stateful Packet Inspection
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

CHAPITRE 1

INTRODUCTION

1.1 De l'évolution des réseaux

Durant toute l'année 2013, Cisco [9] estime que 2,5 milliards d'êtres humains se sont connectés à Internet. Cela signifie donc que plus d'un tiers de l'humanité est utilisateur du réseau de réseaux. On estime qu'à l'horizon de 2018, ce sera plus de la moitié de l'humanité avec 3,9 milliards d'utilisateurs de par le monde. Cet élargissement du réseau vers de nouveaux utilisateurs s'accompagne aussi par l'amélioration des bandes passantes des branches déjà existantes. Par exemple, la société étasunienne Verizon, en charge de la gestion d'une partie des dorsales Internet (*Internet backbone*) et possédant un réseau de plus de 1.287.000 kilomètres utilise les dernières technologies disponibles lors de l'installation de nouveaux câbles soit 40 Gbps pour ses liens sous-marins et 100 Gbps pour ses liens terrestres [62]. Tous ces facteurs ont pour conséquence une croissance remarquable des quantités de données échangées : passant de 51,2 Exaoctets (Exa = 10^{18}) par mois en 2013 à probablement 131,6 Eo en 2018, toujours selon Cisco [9].

Notre utilisation du réseau Internet a considérablement évolué avec les capacités de celui-ci. Les particuliers comme les entreprises se convertissent aux *cloud computing* permettant, par exemple, de déléguer les tâches de stockage ou de calcul à d'autres entités sur le réseau. De la même manière, il faut noter l'utilisation croissante de plateformes dédiées au stockage et à la distribution de fichiers vidéo. Le trafic lié à la plateforme Youtube, par exemple, représenterait à lui seul plus de 5% de la bande passante mondiale en 2011 [14]. La base de données de ce géant de l'Internet augmente de 100 heures chaque minute et dans le même intervalle de temps l'équivalent de 100.000 heures de vidéo est consulté [67]. Cisco [9] estime que la vidéo représentait 57% du trafic Internet en 2013 et que ce nombre passera à 75% en 2018 avec l'accroissement du nombre d'utilisateurs et l'augmentation de la taille des fichiers vidéo utilisés.

Dans ce mémoire, nous présenterons une méthode de référencement dédiée aux fichiers vidéo ainsi qu'un système de détection à grande vitesse de fichiers et de flux connus sur un réseau. Le référencement permet de déterminer les informations à extraire d'un fichier pour que celui-ci puisse être détecté plus tard. Le système pourra analyser les données circulant sur un lien réseau afin de déterminer s'il s'agit de contenus connus ou si ces messages font partie d'une discussion à surveiller.

1.2 Analyse des échanges sur le réseau : raisons et méthodes

Comme nous l'avons mentionné ci-dessus, les progrès de la technologie profitent aux bandes passantes des connexions des utilisateurs de l'Internet. La capacité moyenne de celles-ci s'élevait à 16 Mb/s en 2013 et devrait passer à 42 Mb/s en 2018 [9]. La loi de Nielsen (nommée après Jakob Nielsen) illustre cette évolution : elle stipule que la bande-passante maximale des particuliers croit de 50% par année. Parallèlement, les systèmes de stockage informatiques voient eux aussi leurs capacités décuplées (suivant la loi de Kryder, Mark). Cette croissance se traduit notamment par une réduction considérable du prix moyen par unité de stockage. Malheureusement, ces progrès ne servent pas toujours de nobles desseins.

1.2.1 Exemples de pratiques et de comportements illicites sur Internet

Mentionnons dans un premier temps la diffusion illégale de contenus protégés par le droit d'auteur sur Internet, aussi appelée "piratage". En 2011, il a été estimé que l'échange de ce genre de contenu constituait 23,76% du trafic Internet mondial [14]. En analysant les fichiers échangés, on remarque que les secteurs du divertissement sont particulièrement touchés. La *Motion Pictures Association of America*, par exemple, a annoncé que le manque à gagner pour l'industrie étasunienne du film s'élevait à 20,5 milliards de dollars par année soit 140.000 emplois "perdus" [3]. La *Recording Industry Association of America* estime quant à elle que l'impact du "piratage" sur l'industrie de la musique aux États-Unis s'élève à une perte de 12,5 milliards de dollars par année soit 70.000 emplois [45].

D'autres contenus et comportements, plus malveillants ceux-là, circulent eux aussi sur les réseaux. Ils peuvent être simplement indésirables, comme les pourriels (ou *spam* en anglais) mais d'autres, comme les virus informatiques, peuvent aller jusqu'à mettre à mal l'intégrité de systèmes. Mentionnons aussi l'intrusion dans un système informatique connecté à Internet en vue d'en obtenir des informations confidentielles qui constitue évidemment un délit dans de nombreux pays. Ces "cyber-attaques", pour certaines très médiatisées, peuvent prendre une ampleur titanesque voire même bousculer la géopolitique internationale [7].

Enfin, le partage et la possession de fichiers à contenu pédo-pornographique sont presque unanimement considérés comme illégaux. Or, le développement d'Internet (nombre d'internautes et qualité des équipements) a rendu ce genre de contenu plus courant et plus accessible que jamais [65]. L'ampleur du problème est telle que de nombreux pays ont constitué des brigades de police spécialisées dans la lutte contre la prolifération de la pédo-pornographie sur Internet [16, 38]. Elles traquent les personnes à l'origine des fichiers, les fournisseurs (sites) et

les consommateurs (internautes) et parviennent à récupérer, sur des équipements saisis lors d'enquêtes, les objets du délit. Ces bases de données de fichiers illicites peuvent être utilisées par les forces de police pour de futures enquêtes.

1.2.2 Détection de contenus potentiellement illégaux

Sur un nœud du réseau

Internet est un réseau de réseaux interconnectés basé sur la commutation de paquets. Chaque nœud peut être connecté à de nombreux autres et les paquets peuvent potentiellement emprunter des chemins différents pour atteindre un même point. Cette dernière caractéristique constitue un inconvénient pour la détection du contenu de ces paquets mais il ne s'agit pas du seul. Nous aurons l'occasion de revenir plus en détail sur les principaux protocoles réseaux par la suite mais, pour l'heure, on peut mentionner quelques caractéristiques des réseaux affectant le transfert d'un fichier entre deux points et qui ne faciliteront pas notre travail :

- Le fichier peut être segmenté en paquets de tailles non-définies.
- Les paquets peuvent être transmis dans n'importe quel ordre.
- L'ordre dans lequel ils sont reçus peut différer de l'ordre dans lequel ils ont été transmis.
- Tous les paquets ne transitent pas nécessairement par les mêmes nœuds.
- Des paquets peuvent être perdus.

Ce genre de contraintes suggère que la reconstitution complète des fichiers à partir des paquets capturés depuis un seul point du réseau est peu probable. C'est d'autant plus improbable si l'on considère que les flux de millions d'internautes transitent et se mélangent sur les mêmes nœuds. D'autre part, notons que la reconstitution de tels fichiers, même temporaire, peut être illégale.

En observant les entêtes des paquets transitant par un point du réseau, on peut simplement extraire leurs adresses de source et de destination. Un filtrage simple consiste à comparer ces adresses à celle des sites (point du réseau) dont l'accès doit être bloqué. Cette pratique est très courante, notamment dans le cadre de sites diffusant du contenu pédo-pornographique [35]. En revanche, la méthode est limitée puisqu'elle nécessite de connaitre à l'avance l'adresse de ces sites. Il est, de plus, relativement aisé pour un internaute de contourner le nœud du réseau qui applique de tels filtrages ou, pour le site, d'opérer sous différentes adresses.

Une méthode plus avancée consiste à analyser les données incluses dans le paquet afin de déterminer si ceux-ci doivent être signalés, réorientés, ou simplement supprimés. On parle alors de DPI (*Deep Packet Inspection*). Cela nécessite d'avoir une base de données de réfé-

rences auxquelles sont comparées les données des paquets. On distingue alors le référencement, la tâche qui consiste à analyser les fichiers qui devront être identifiés à nouveau lors de la détection, étape à laquelle on compare le contenu des paquets à notre base de données de références. Nous reviendrons sur cette méthode plus tard dans ce mémoire.

Avec accès total

La situation change lorsque l'on peut, légalement et techniquement, avoir accès à la totalité du flux ou du fichier. La société étasunienne Google, par exemple, lutte contre la diffusion de contenus protégés par le droit d'auteur sur ses services (transitant donc par ses serveurs). Ainsi le système ContentID [21] analyse les flux vidéos de la plateforme de partage de vidéos Youtube et du service *Hangout*. Il est annoncé que plus de 25 millions de fichiers sont référencés et que le système traite l'équivalent de 400 années de vidéos par jour pour plus de 5000 ayant-droits [67].

Enfin, la situation se simplifie aussi lorsque l'on souhaite détecter la présence de fichiers ou de segments de fichiers sur un support de stockage. Dans le domaine judiciaire, par exemple, les disques-durs saisis lors d'enquêtes sont intégralement analysés pour y retrouver certains fichiers (connus ou non). Le processus de tri dans les fichiers peut être grandement accéléré grâce au hachage, sur lequel nous reviendrons, et grâce à la base de données du NIST [41] qui référence une collection de fichiers courants et légaux (donc inintéressants pour un enquêteur).

1.3 Cahier des charges de notre système et objectifs de recherche

Des observations précédentes on détermine quelques critères que doit suivre un système de détection de fichiers connus en un point du réseau pour être efficace. Tout d'abord, concernant le référencement, notre système doit être capable d'analyser des fichiers afin d'en extraire les informations adéquates pour pouvoir, plus tard, détecter des segments de ces fichiers disséminés dans des paquets réseau. D'autre part, les transferts de fichiers vidéo pouvant être liés à la distribution illégale de contenus protégés par le droit d'auteur ou de contenus à caractère pédo-pornographique, il conviendrait aussi de proposer une méthode de référencement dédiée à ce type de fichier. On note donc le premier objectif de recherche :

Objectif 1 : Proposer une méthode d'extraction d'empreintes de fichiers vidéo qui garantisse l'originalité de ces empreintes et qui faciliterait la détection de segments de ces fichiers disséminés dans des paquets réseau.

Concernant la tâche de détection, notre système se baserait sur les travaux de Jonas Lerebours [29] qui propose un système de détection de fichiers connus à grande vitesse sur GPU. Ce système devra être capable de prendre en charge les empreintes préalablement calculées lors du référencement. Nous souhaiterions pouvoir détecter, en plus des paquets ayant du contenu connu, ceux qui appartiendraient à un flux de paquets qui auraient déjà levé des alertes ou qui comporteraient une adresse à surveiller. Enfin, notre système devra pouvoir être utilisé par des gestionnaires de larges réseaux (bandes passantes importantes et nombreux usagers) tels que les grandes entreprises, les Fournisseurs d'Accès à Internet (FAI) ou encore les gestionnaires des dorsales Internet. Cela reviendrait donc à pouvoir analyser les flux provenant de liens à 40 Gb/s voire 100 Gb/s. On note donc :

Objectif 2 : Proposer un système de détection de segments de fichiers connus et de flux réseaux connus capable de traiter les paquets circulant sur des liens réseaux à très haute vitesse (40 Gb/s et plus).

1.4 Plan du mémoire

Le présent mémoire sera découpé comme suit. Dans un premier temps, nous détaillerons le contexte des réseaux et signalerons les différentes solutions existant dans le domaine de la détection de données connues afin de valider les méthodes que nous emploierons dans notre système. Lors du troisième chapitre, nous présenterons notre démarche pour remplir le premier objectif de recherche : le référencement dédié aux fichiers vidéo. Nous prendrons pour exemple un des formats vidéos les plus utilisés aujourd'hui : le MP4 avec l'encodeur H.264. Le quatrième chapitre sera dédié au second objectif de recherche. Nous y mentionnerons d'abord les choix effectués pour la conception du système de détection, principalement implémenté sur GPU, puis nous présenterons les résultats et les performances mesurées de celui-ci. Enfin, le chapitre cinq clôturera ce mémoire et proposera des pistes pour la poursuite des travaux qui auront été présentés.

CHAPITRE 2

REVUE DE LITTÉRATURE

Ce mémoire vise à proposer un système de détection de fragments de fichiers connus et de flux suspects sur un lien 40 Gb/s. De plus, nous nous concentrerons sur le référencement de fichiers vidéo. Ce chapitre de revue de littérature a pour but de présenter les différents systèmes, algorithmes et matériels déjà existants ce qui nous permettra de justifier les choix de conception que nous ferons par la suite.

2.1 Les protocoles des réseaux informatiques

Il convient dans un premier temps de présenter et de détailler les protocoles auxquels il sera fait référence dans ce mémoire. Ceux-ci peuvent tout d'abord être introduits comme faisant partie du modèle OSI (*Open Systems Interconnection*) représenté au tableau 2.1. Cette vision en couches des fonctions de communication met en évidence les notions d'abstraction et d'encapsulation qui existent entre celles-ci. À titre d'exemple, deux protocoles sont indiqués pour chaque couche. Il ne s'agit pas d'une liste exhaustive. Nous allons porter une attention particulière à ceux figurant en caractères gras, les plus utilisés pour les couches 2, 3 et 4. Pour cela, nous procèderons dans l'ordre de désencapsulation, c'est à dire de la couche Liaison (2) à la couche Transport (4).

2.1.1 Couche Liaison : Ethernet

Ethernet est un protocole pour les réseaux locaux filaires (LAN) occupant les couches Physique et Liaison du modèle OSI. Il permet d'acheminer des données entre deux points qui partagent un même canal. Ces deux points sont identifiés par leurs adresses MAC (*Media*

Tableau 2.1 Modèle OSI

N°	Nom	Exemples de protocole
7	Application	HTTP, FTP
6	Présentation	MPEG, JPEG
5	Session	SCP, SQL
4	Transport	**TCP**, **UDP**
3	Réseau	**IP**, ICMP
2	Liaison	**Ethernet**, Token Ring
1	Physique	

Access Control), de six octets chacune. Il s'agit en général d'adresses matérielles fixées par le fabricant de l'interface réseau, et elles sont considérées comme uniques. Les données sont transmises sous la forme d'une trame Ethernet, schématisée par la figure 2.1.

On peut voir que les paquets Ethernet consistent d'un entête de 14 octets suivi par un champ de taille variable contenant les données. On notera aussi la présence d'une signature après les données. Dans l'en-tête, on distingue trois champs :

- L'adresse MAC de l'expéditeur occupant six octets.
- L'adresse MAC du destinataire, six octets elle aussi.
- Le champ "Taille ou type", aussi appelé "Ethertype", a un rôle double en fonction de sa valeur (codée sur deux octets) :
  - Si sa valeur est inférieure ou égale à 1500 alors il s'agit bel et bien de la taille en octets de la trame courante (les trames Ethernet I, n'étant plus utilisées aujourd'hui).
  - Si supérieure à 1500, alors la valeur permet de spécifier le protocole de la couche Réseau inclus dans le champ de données (voir tableau 2.2).

Le champ de données occupe entre 46 et 1500 octets. Dans le cas où les informations à transmettre sont plus larges, elles devront être fragmentées en plusieurs trames distinctes avant de pouvoir être transmises sur le canal. Ce champ pourra lui-même contenir les données propres aux protocoles utilisés dans les couches supérieures du modèle OSI.

Enfin, le champ CRC-32 contient la signature, sur quatre octets, de l'en-tête et des données. Cette signature permet de vérifier l'intégrité du paquet : si la signature ne correspond pas au paquet celui-ci est intégralement jeté. Nous reviendrons plus tard sur la manière dont est générée cette signature lorsque nous aborderons le hachage à la section 2.3.2. La taille d'un paquet Ethernet est donc comprise entre 64 et 1518 octets.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Adresse MAC source																															
Adresse MAC source (suite)																Adresse MAC destination															
Adresse MAC destination (suite)																															
Taille ou type																Données (entre 46 et 1500 octets)															
Données (suite)																															
Signature CRC-32																															

Figure 2.1 Trame Ethernet

Tableau 2.2 Exemples de valeurs Ethertype et protocoles correspondants

Valeurs	Protocoles
0x0800	**IPv4**, ICMP
0x86DD	IPv6
0x0842	Wake-on-LAN
0x8035	RARP
0x809B	AppleTalk

2.1.2 Couche Réseau : le protocole IP

Internet Protocol (IP) permet de palier à certaines limitations de Ethernet : les deux interlocuteurs n'ont pas nécessairement à faire partie du même (sous-)réseau. La source et le destinataire du message sont, cette fois-ci, définis par des adresses logicielles. Les valeurs de ces adresses détermineront le routage du message qui n'est pas fixé à l'avance : on dit que IP n'est pas "orienté connexion", contrairement par exemple au réseau téléphonique. Par conséquent, des paquets IP émis dans un certain ordre ne sont pas assurés d'arriver dans cet ordre, ni même d'arriver tout court. Ces inconvénients seront adressés dans la couche Transport, notamment par le protocole TCP (section 2.1.3).

Nous nous concentrerons dans ce mémoire sur la version 4 de IP. Elle est en effet la plus répandue aujourd'hui mais est limitée par le nombre d'adresses qu'elle peut supporter. Étant donné qu'une adresse IPv4 est codée sur 4 octets, on peut déterminer un maximum de $2^{4\times8}$ adresses, soit un peu plus de 4 milliards. La version 4 de IP sera par conséquent peu à peu abandonnée et remplacée par la version 6 qui utilise, quant à elle, des adresses de 16 octets (soient $3,4 \times 10^{38}$ adresses possibles).

La configuration d'une trame IPv4 est présentée dans la figure 2.2. On notera quelques champs importants dans son entête :

- La version de IP utilisée.
- La taille de l'entête et la taille totale de l'entête *et* des données.
- Le protocole inclus dans le champs de données. Par exemple, si la valeur contenue dans le champ vaut 6 ou 17, cela indique respectivement TCP ou UDP.
- Les adresses IP de la source et du destinataire, codée sur 4 octets chacune.

2.1.3 Couche Transport : protocoles TCP et UDP

Ces deux protocoles de la couche Transport utilisent la notion de "port" logiciel. Sur demande au du système d'exploitation, une application peut se voir attribuer un port au travers

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td colspan="4">Version</td><td colspan="4">Taille entête</td><td colspan="6">Service</td><td colspan="2">Cong.</td><td colspan="16">Taille totale</td></tr>
<tr><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Offset du fragment</td></tr>
<tr><td colspan="8">Durée de vie</td><td colspan="8">Protocole inclus</td><td colspan="16">Signature de l'en-tête</td></tr>
<tr><td colspan="32">Adresse IP source</td></tr>
<tr><td colspan="32">Adresse IP destination</td></tr>
<tr><td colspan="32">Options (facultatives)</td></tr>
<tr><td colspan="32">Données</td></tr>
</table>

Figure 2.2 Trame IPv4

duquel elle pourra accéder aux interfaces réseau pour envoyer ou recevoir des données. Étant codé sur 2 octets, on a donc théoriquement $2^{2\times 8} = 65536$ ports à notre disposition. Dans les faits, certains ports sont réservés.

TCP et UDP ont donc en commun le fait de spécifier deux ports : un port pour la source et un port pour le destinataire du message. Néanmoins, les deux protocoles ne sont pas redondants et offrent des services bien différents. TCP est "orienté connexion" : deux machines doivent procéder à une "poignée de main" avant de s'envoyer des messages qui seront scrupuleusement acquittés. Ce protocole permet donc de palier à certaines limitations de IP et de Ethernet qui, jusque là, ne permettaient pas de s'assurer de la bonne délivrance des messages. UDP, en revanche, ne propose rien de plus que IP si ce n'est bien sûr de spécifier des ports logiciels. Les messages ne sont pas assurés d'arriver dans l'ordre, ni même d'arriver à destination. Ce qui pourrait apparaitre comme un défaut peut être préférable pour certains cas d'utilisation dans lesquels on ne peut se permettre le temps d'établir les connexions ("poignées de main") ou d'acquitter chacun des messages (voix sur IP, *streaming* vidéo et jeux-vidéos par exemple).

Les entêtes TCP et UDP sont représentés sur les figures 2.3 et 2.4. Compte-tenu des services offerts par TCP, il n'est pas étonnant de constater que son entête occupe près de quatre fois l'espace de UDP. Pour ce dernier, on remarquera notamment les champs suivants :

– Ports source et destination, occupant deux octets chacun.
– Taille totale de l'entête et des données (la taille de l'entête étant fixe).

Quant à TCP, on notera en particulier :

– Les ports source et destination.
– La taille de l'entête, variable du fait des options facultatives (comme pour IPv4).

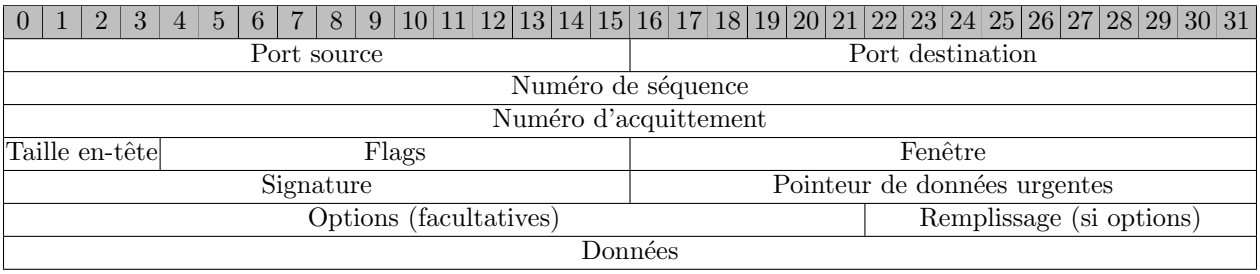

Figure 2.3 Trame TCP

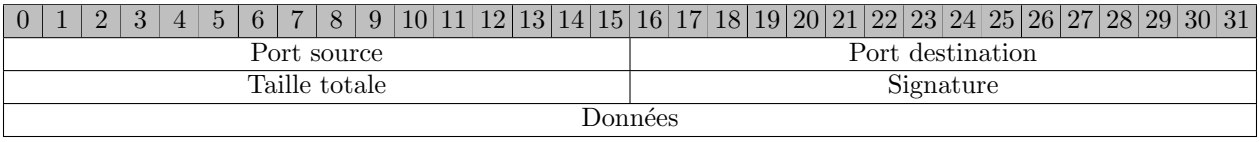

Figure 2.4 Trame UDP

2.2 Systèmes de détection d'intrusion

2.2.1 Principe

Maintenant que nous savons ce à quoi ressemble un message circulant sur un réseau, nous pouvons traiter des systèmes qui les analysent à des fins de sécurité. Certains messages (ou certaines suites de messages) peuvent en effet mettre en péril l'intégrité de systèmes informatiques ou encore révéler des données confidentielles. Ces "attaques" ou "intrusions" peuvent être identifiées voire bloquées par ce que nous appelons des IDS (*Intrusion Detection System*) et des IPS (*Intrusion Prevention System*). Cette identification peut être basée sur la lecture des entêtes des paquets, on parle alors de SPI (*Stateful Packet Inspection*), ou sur le contenu de leur champ de données, il s'agit là de DPI (*Deep Packet Inspection*).

Ces systèmes existent sous différentes catégories. On peut par exemple distinguer les systèmes basés sur le réseau de ceux basés sur l'hôte [17]. Les premiers ont accès à tous les messages circulant dans un réseau, les seconds n'accèdent qu'aux données concernant l'hôte sur lequel ils sont installés. Ces deux catégories ont chacune des avantages et des inconvénients qui leurs sont propres, si bien qu'on trouve aujourd'hui beaucoup de systèmes situés à la fois au niveau du réseau et à celui de l'hôte.

Nous allons sommairement présenter les deux types de détection qui existent : la détection d'anomalies et la détection d'abus [25]. Ici encore, les solutions commerciales ont tendance à concilier les deux méthodes pour profiter des avantages de chacune.

2.2.2 Détection d'anomalies

Les "anomalies", dont nous parlons ici, sont des écarts par rapport à un comportement considéré comme habituel. Cela sous-entend évidemment d'avoir défini ce qu'est un profil "normal" d'activité. Un comportement inhabituel peut alors générer une alerte et le système pourra éventuellement s'en protéger. Ce type de détection a recours à des méthodes d'**apprentissage artificiel**. Sans entrer plus dans les détails, on mentionnera que différentes approches existent comme les apprentissages "supervisés" et "non-supervisés", respectivement illustrés dans le domaine de la détection d'intrusions par [20] et [43].

L'avantage principal de la détection d'anomalies est le fait de pouvoir détecter des attaques jusqu'alors inconnues. Le taux de faux-positifs (considérer à tort qu'un comportement anodin représente une attaque) lors de la détection est en revanche généralement assez élevé.

2.2.3 Détection d'abus

Un abus est l'occurrence d'un comportement ou d'un évènement préalablement déterminé comme une attaque. Il s'agit de détecter un motif, *pattern* en anglais, qui est la "signature" d'un type d'attaque connu. Le système comporte donc une base de données de signatures, de motifs, qu'il essaye de retrouver parmi les messages visibles sur le réseau. `Snort` [48] est l'exemple typique dans le domaine des IDS basés sur les signatures (abus) même si aujourd'hui, comme mentionné plus haut, celui-ci s'étend aussi à la détection d'anomalies.

Ce genre de détection est très fiable : une attaque dont on connait une signature précise pourra être détectée et les taux de faux-positifs et de faux-négatifs sont en général très faibles. Néanmoins, on ne peut se prémunir contre un nouveau type d'attaque dont la signature est encore inconnue. Dans la section suivante, nous présenterons les différentes approches pour la détection de données connues ou de leur signature. Certaines d'entre elles sont fréquemment employées dans des IDS/IPS basés sur la détection d'abus.

2.3 Détection de données connues

Il existe de nombreuses méthodes pour la reconnaissance de fichiers, de morceaux de fichiers, ou de motifs. Nous allons procéder de l'approche la plus naïve pour en arriver à l'algorithme de *max-Hashing*, l'un des piliers du système proposé dans ce mémoire.

2.3.1 Comparaison de chaînes de caractères

Nous parlerons en effet ici de chaînes de caractères du fait que les algorithmes présentés ci-après sont utilisés notamment pour la détection de plagiat, de pourriel (*spam* en anglais), de séquences de gènes d'ADN, voire même dans le domaine des moteurs de recherche. Cela rendra aussi la lecture plus agréable puisque l'on peut substituer les valeurs hexadécimales d'octets par leur représentation dans l'alphabet latin selon le codage ASCII.

Nous souhaitons trouver dans un texte représenté par une chaîne de caractères S_1 de taille k_1, un mot ou une phrase S_2 de taille k_2. On cherche donc les m occurrences de S_2 dans S_1 avec évidemment $k_1 \geq k_2$. Pour l'illustration, on considérera que S_1 = "ababaabcabdabcac" et S_2 = "abcac". On note que S_2 se retrouve une seule fois à la fin de S_1 ($m = 1$).

Comparaison exacte, naïve et améliorée

La première approche, dite naïve, consiste à comparer S_2 à toutes les sous-chaînes de k_2 caractères de S_1. On commence par aligner le premier caractère de S_2 au premier caractère de S_1. À la première différence on "décale" S_2 : on la compare à S_1 à partir de son second caractère comme le montre la figure 2.5. Cette méthode simple et facile à implémenter est néanmoins très peu optimisée (voir tableau 2.3). On ne garde en effet aucune information sur les comparaisons qui ont déjà eu lieu aux itérations précédentes. Sur la figure 2.5 par exemple, à l'itération 2, on compare le premier caractère de S_2 ("a") au premier de S_1 ("b"). Or ce dernier a déjà été lu précédemment, lors de l'itération 1, mais aucune information sur cette lecture et cette comparaison n'a été conservée.

Itérations	a	b	a	b	a	a	b	c	a	b	d	a	b	c	a	c
1	a	b	**c**	*a*	*c*											
	=	=	≠													
2		**a**	*b*	*c*	*a*	*c*										
		≠														
3			a	b	**c**	*a*	*c*									
			=	=	≠											
4				**a**	*b*	*c*	*a*	*c*								
				≠												
...								...								
12												a	b	c	a	c
												=	=	=	=	=

Figure 2.5 Approche naïve pour la comparaison de chaînes de caractères

La première amélioration, proposée par Knuth, Pratt et Morris [26] en 1977, consiste à réaliser un pré-traitement sur S_2 avant même de lire S_1. Il s'agit de tirer parti de la configuration des caractères de S_2 voire même de leurs redondances. Dans S_2 = "abcac", il y a un "a" en première et en quatrième position. Or, dans le cas où une comparaison échouerais seulement pour le cinquième caractère de S_2, "c", on sait que le dernier caractère commun de S_1 et S_2 était un "a" et qu'il y en a un au début de S_2 suivi d'un caractère différent de "c". Par conséquent, on peut directement décaler S_2 afin de repartir du "a" de S_1 sans pour autant le vérifier à nouveau (itérations 4 et 5 de la figure 2.6).

L'approche proposée par Boyer et Moore [5], et qui sera plus tard améliorée par Horspool [19], permet entre autres de prendre en compte l'occurrence, dans S_1, de caractères n'appartenant pas à l'alphabet du motif S_2. Dans notre exemple, le caractère "d" apparait dans S_1 (onzième position) mais pas dans S_2. On peut donc, une fois qu'il a été lu (c'est-à-dire comparé à un caractère de S_2), l'ignorer complètement par la suite en faisant glisser S_2 jusqu'à le dépasser. Sur la figure 2.6, cela reviendrait à ne pas effectuer l'itération 6. Néanmoins si la taille a_1 du dictionnaire de S_1 est importante, cela alourdi le pré-traitement.

Enfin, l'algorithme Rabin-Karp [24] diffère enormément de ceux présentés précédemment en cela qu'il utilise une **fonction de hachage**. Une **empreinte** est générée à partir du motif à rechercher S_2 et sera comparée à celles générées sur S_1. A chaque itération, il n'y a donc qu'une seule comparaison à effectuer mais qui nécessite la génération de l'empreinte sur S_2. Une illustration est donnée sur la figure 2.7 où $E = F(S_2[0; k_2 - 1])$ est l'empreinte de S_2 générée par la fonction de hachage F et $e_n = F(S_1[n; n + k_2 - 1])$ l'empreinte générée à partir du $n^{\text{ième}}$ caractère de S_1. Plus d'informations sont fournies concernant les fonctions de hachage à la section 4.4.4.

Expressions rationnelles (régulières)

Les expressions rationnelles présentent une théorie générale pour la recherche de motifs. Elle inclut la recherche exacte de motifs, comme précédemment illustrée, mais aussi plus largement la recherche de motifs complexes. Ces derniers peuvent représenter une multitude, voire une infinité, de motifs "exacts". Par exemple, selon la syntaxe généralement employée, l'expression rationnelle "[ab]+cac" englobe S_2 = "abcac" mais aussi "ababcac", "abababcac", "ababababcac", et une infinité d'autres motifs ayant la même construction (le motif "ab" peut être répété n fois, avec $n > 0$).

Itérations	a	b	a	b	a	a	b	c	a	b	d	a	b	c	a	c
1	a	b	**c**	*a*	*c*											
	=	=	≠													
2		a	b	**c**	*a*	*c*										
		=	=	≠												
3			a	**b**	*c*	*a*	*c*									
			=	≠												
4				a	b	c	a	**c**								
				=	=	=	=	≠								
5							*a*	b	**c**	*a*	*c*					
								=	≠							
6									**a**	*b*	*c*	*a*	*c*			
									≠							
7												a	b	c	a	c
												=	=	=	=	=

Figure 2.6 Application de l'algorithme de Knuth-Pratt-Morris

Itérations	a	b	a	b	a	a	b	c	a	b	d	a	b	c	a	c
1	$e_0 \neq E$															
2		$e_1 \neq E$														
3			$e_2 \neq E$													
4				$e_3 \neq E$												
...								...								
12												$e_{11} = E$				

Figure 2.7 Application de l'algorithme de Rabin-Karp

C'est notamment à Rabin et Scott, en 1959 [44], que l'on attribue l'adaptation des expressions rationnelles en machines à états (automates) finis. Ces dernières, déterministes ou non, peuvent être construites selon une multitude de méthodes comme par exemple celle de Thompson implémentée dans le logiciel pour Unix `grep` [58], et qui inclura aussi la méthode de Aho et Corasick [2]. Cette dernière est la plus fréquemment utilisée dans les IDS (comme `Snort` [48]) et les logiciels antivirus (comme `ClamAV` [10]) notamment en raison de sa vitesse d'exécution (voir 2.3).

Reconnaissance approximative

Pour compléter notre rapide tour d'horizon, il faut mentionner le domaine de la reconnaissance approximative des chaines de caractères. Des algorithmes permettent de détecter toute occurrence d'une chaine de caractère jugée suffisamment "proche" du motif à rechercher ; la distance entre deux chaines se mesurant au nombre de différences qui existent entre elles. Il peut s'agir d'une insertion (rajout d'un caractère), d'une suppression, ou d'une substitution (remplacement d'un caractère par un autre). Navarro [36] dresse un état de l'art des tendances dans le domaine, le lecteur intéressé pourra s'y référer.

Bilan sur la reconnaissance de chaines de caractères

Afin de conclure sur la reconnaissance de chaînes de caractères, il convient d'indiquer la raison pour laquelle, malgré les avantages et la puissances de certains algorithmes, ceux-ci ne conviendraient pas pour la détection de millions d'objets. Tout d'abord, il faut mentionner que chaque nouvelle règle et nouvelle signature à détecter alourdit considérablement le processus de détection. D'autre part, la possession et le stockage de ces objets peuvent être illégaux, et ce, qu'ils soient complets ou non. Or, les algorithmes présentés plus haut requièrent le stockage du fichier à détecter ou d'une partie de celui-ci. Néanmoins, l'algorithme de Rabin-Karp [24] indique une piste intéressante : le recours aux fonctions de hachage qui permettent de cacher le fichier original, grâce à l'injectivité de celles-ci, et de réduire la taille de sa signature.

Tableau 2.3 Complexité des algorithmes de comparaison de chaînes de caractères

Algorithme	Pré-traitement	Pire cas	Meilleur cas
Approche naïve	*aucun*	$O(k_1 \times k_2)$	$O(k_1)$
Knuth-Pratt-Morris	$O(k_2)$	$O(k_1 \times k_2)$	$O(k_1)$
Boyer-Moore	$O(k_2 + a_1)$	$O(k_1 \times k_2)$	$O(k_1/k_2)$
Rabin-Karp	$O(k_2)$	$O(k_1 \times k_2)$	$O(k_1 + k_2)$
Aho-Corasick	$O(k_2)$	$O(k_1 + m)$	$O(k_1)$

2.3.2 Hacher pour représenter les données

Le recours à une fonction de hachage peut avoir plusieurs intérêts : encoder un message afin de le rendre illisible et/ou le réduire afin d'obtenir son empreinte ou sa signature. Ces caractéristiques sont intéressantes lorsqu'on ne peut pas stocker des millions de fichiers de manière lisible, techniquement de par leur taille et légalement de par leur nature. On notera, avec F la fonction de hachage et A et B deux champs de données, que $A = B \rightarrow F(A) = F(B)$. Réciproquement, on veut que $F(A) = F(B)$ indique avec le maximum de probabilité que $A = B$. L'incertitude sur ce dernier point est due à la taille des empreintes qui sont généralement plus petites que les données sur lesquelles elles ont été calculées. Cette section présente une brève revue des méthodes de hachage.

Empreinte numérique de fichier/message

Une des premières applications des fonctions de hachage mentionnées dans ce mémoire fut la vérification de l'intégrité d'un message Ethernet à la section 2.1.1 avec le CRC-32. Le principe ici est de calculer une empreinte de taille fixe (32 bits) à partir du message que l'on souhaite envoyer et de joindre cette empreinte au paquet. Le récepteur du paquet peut alors vérifier si le message qu'il a reçu génère bien la même empreinte. Si ce n'est pas le cas, alors il est certain que le message a été altéré lors de sa transmission et doit donc être jeté.

Dans l'algorithme MD5 (*Message Digest 5*, [47]), l'empreinte générée par la fonction de hachage compte 128 bits. MD5 est généralement utilisé pour vérifier le transfert de fichiers de grande taille. L'empreinte est générée puis fournie par la source du fichier afin que le destinataire puisse vérifier que le transfert du fichier s'est bien déroulé. Là encore, si une différence est observée entre l'empreinte fournie par la source est celle calculée par le destinataire, il convient de transférer à nouveau le fichier. Cet algorithme a aussi été employé afin de stocker les mots de passe de manière plus sécuritaire. Néanmoins, MD5 a très rapidement montré ses limitations : il est devenu très facile et rapide à "attaquer" [66]. C'est ce qui a motivé son remplacement par SHA-1 (*Secure Hash Algorithm-1*, [13]) dont les empreintes sont codées sur 160 bits, même si ce dernier a déjà montré quelques signes de faiblesse [64].

Hachage après fragmentation

Dans le scénario de transfert de fichier présenté ci-dessus, si la signature indiquée par la source est différente de celle générée par le récepteur, tout le fichier est jeté et devra être transféré à nouveau. Ce défaut devient particulièrement pénalisant dans le cas de fichiers volumineux et, par conséquent, longs à échanger. Afin de parer à ce problème, Merkle [34]

propose de fragmenter le fichier en plusieurs segments et de hacher ceux-ci indépendamment les uns des autres. Les empreintes ainsi générées sont ensuite elles-mêmes hachées deux-à-deux afin de constituer un arbre d'empreintes. Un arbre de Merkle à partir de quatre fragments est illustré à la figure 2.8. Si l'empreinte finale $E_{1,2,3,4}$ n'est pas reconnue, on peut alors remonter dans les empreintes afin de déterminer quelle(s) partie(s) du fichier nécessite(nt) d'être transférée(s) à nouveau.

Il est à noter que le hachage de segments de fichiers est aussi très fortement utilisé dans le domaine judiciaire afin de retrouver, sur du matériel saisi lors d'enquêtes, des traces de fichiers illégaux. Harbour a présenté le logiciel `dcfldd` [18] pour ce genre d'utilisation. Le logiciel calcule les signatures des secteurs du disque dur et les compare à sa propre base de données de signatures. Lorsque stocké sur un disque dur, un fichier est en effet segmenté puis placé dans des secteurs de 4 ko. L'un des avantages de cette méthode est d'être capable de retrouver des fragments de fichiers même quand ceux-ci ont été supprimés par le système d'exploitation mais que les secteurs n'ont pas encore été ré-écris.

En revanche, ces méthodes sont beaucoup moins performantes lorsqu'on les applique sur les données contenues dans un paquet Ethernet. Celles-ci sont en effet segmentées de manière à priori imprévisible et l'on ne peut pas stocker les empreintes pour toutes les combinaisons de segmentation possibles. Par extension, ces méthodes sont aussi très sensibles à toute modification des données : des substitutions affecteront les empreintes des segments dans lesquels elles sont appliquées et une simple insertion/suppression aura un impact sur toutes les empreintes calculées à partir de celle-ci. Sur l'exemple de la figure 2.8, si un simple bit est inversé sur la partie p_2 du fichier, il est très probable que l'empreinte E_2 en soit modifiée (et par conséquent les empreintes $E_{1,2}$ et $E_{1,2,3,4}$ aussi). Si en revanche on insère un nouvel octet sur la partie p_2, alors seule E_1 restera intacte.

Hachage de fenêtres délimitées par le contexte

Une solution au problème de décalage des données consisterait à générer les empreintes d'un fichier sur des parties dont les limites sont définies par le contexte même du fichier. Par exemple, dans le cas d'un fichier texte, les fenêtres peuvent être délimitées par un caractère particulier : le point ".". Une phrase connue pourra donc être identifiée même si sa position diffère de celle dans le fichier original. Kornblum [27] présenta cette méthode sous le terme de *Context-Triggered Piecewise Hashing* ou CTPH en généralisant la détermination des bornes des fenêtres à des fichiers non-textuels.

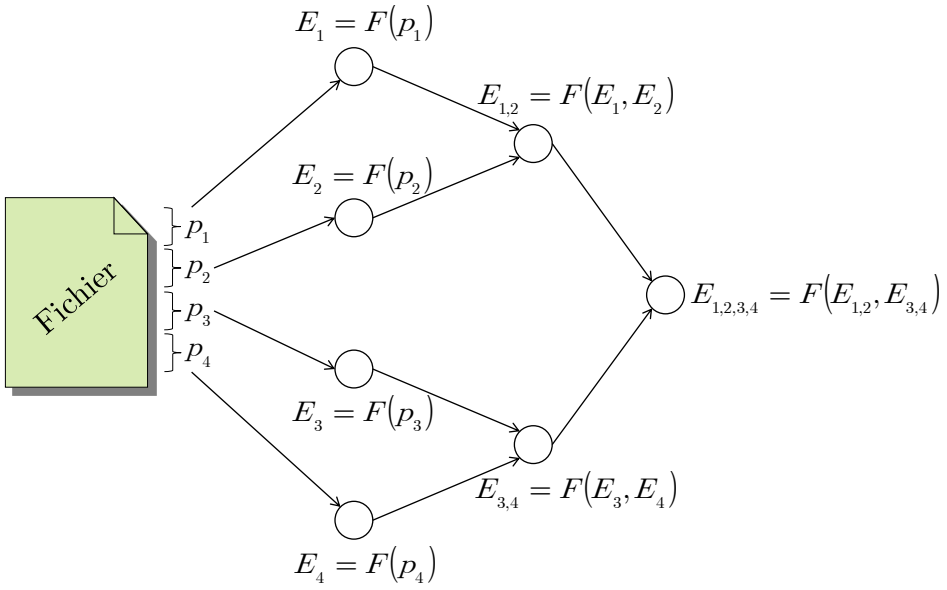

Figure 2.8 Illustration d'un arbre de Merkle sur 4 segments

Il utilise en effet une fenêtre glissant sur les données, très similairement à l'algorithme de Karp-Rabin (section 2.3.1). Pour le CTPH, une fenêtre de taille fixe glisse sur les données octet par octet et génère une clé à chaque itération. Si la clé respecte un certain critère alors on considère cette position comme étant le début (ou la fin) d'un segment intéressant du fichier. Une empreinte pourra ensuite être calculée sur ce segment. Kornblum a implémenté son algorithme dans le logiciel `ssdeep` [28] en se basant sur `Spamsum` de Tridgell [59]. Dans le domaine judiciaire, Roussev a détaillé l'utilisation de cette méthode et d'autres [49, 50, 51].

Algorithme de *winnowing*

Cet algorithme développé par Schleimer *et al.* [53] se base lui aussi sur celui de Rabin-Karp (section 2.3.1). Avec une fenêtre W, de taille fixe S_W, et glissant sur les S_f octets du fichier, on génère $S_f - S_W + 1$ empreintes. L'algorithme suggère alors, pour chaque groupe de n empreintes consécutives, de sélectionner celle qui a la valeur minimale. En cas d'égalité on choisira l'empreinte la plus à droite dans le groupe. Enfin, on constitue la signature du fichier en rassemblant les empreintes sélectionnées précédemment. Si la même empreinte a été choisie dans plusieurs groupes on ne gardera que sa première occurrence. La figure 2.9 illustre l'algorithme de *winnowing* avec $S_W = 16$ octets et $n = 4$.

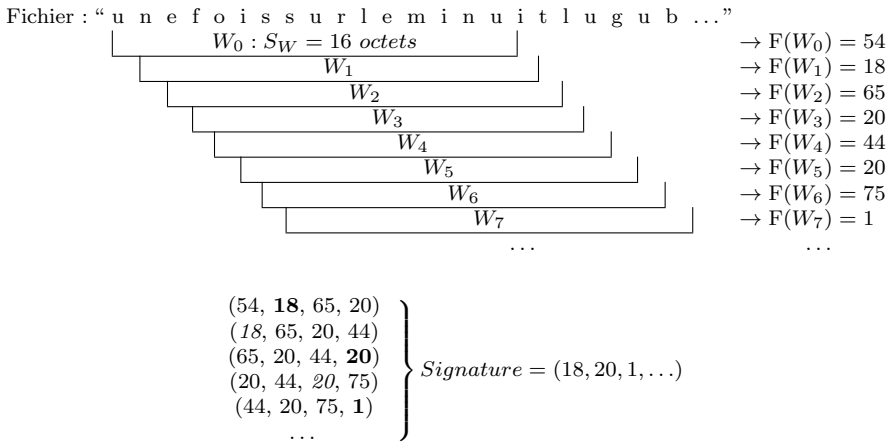

Figure 2.9 Illustration de l'algorithme de *winnowing*

2.3.3 Max-Hashing

L'algorithme de *max-Hashing* a été développé par David [11], dans le but de détecter des fragments de fichiers connus contenus dans des paquets circulant sur un lien réseau à haute vitesse. Comme tous les algorithmes de détection de données connues, il est composé de deux phases distinctes : en amont, le référencement des fichiers ; puis, en aval, la détection des fragments de ceux-ci.

On retrouve un fonctionnement similaire à l'algorithme de *winnowing*, il est illustré à la Figure 2.10. On considère d'abord le fichier segmenté en une multitude de blocs. Lors du référencement, les blocs ont une taille fixe choisie de S_B octets. Lors de la détection, la taille est inconnue : on prend le contenu du champ de données des paquets. Pour chaque bloc, on applique une fonction de hachage F sur toutes les fenêtres W de S_W octets. L'empreinte qui a la valeur la plus grande (le maximum local) est conservée pour représenter ce bloc. L'empreinte est donc calculée sur une petite portion de ce bloc, ce qui constitue un avantage, étant donné que, plus cette portion est petite, plus la probabilité qu'elle soit sectionnée, modifiée ou altérée est faible. La propriété, pour une fenêtre, de générer une empreinte qui a la valeur maximale localement se retrouvera lors de la détection même sans connaitre à l'avance la configuration de la segmentation du fichier en paquets réseau. Tant que la fenêtre elle-même n'est pas tronquée, et on a vu que la probabilité que cela arrive est faible, soit elle conserve la propriété de générer le maximum local soit cette propriété passe à une autre fenêtre, elle aussi référencée comme générant un maximum local.

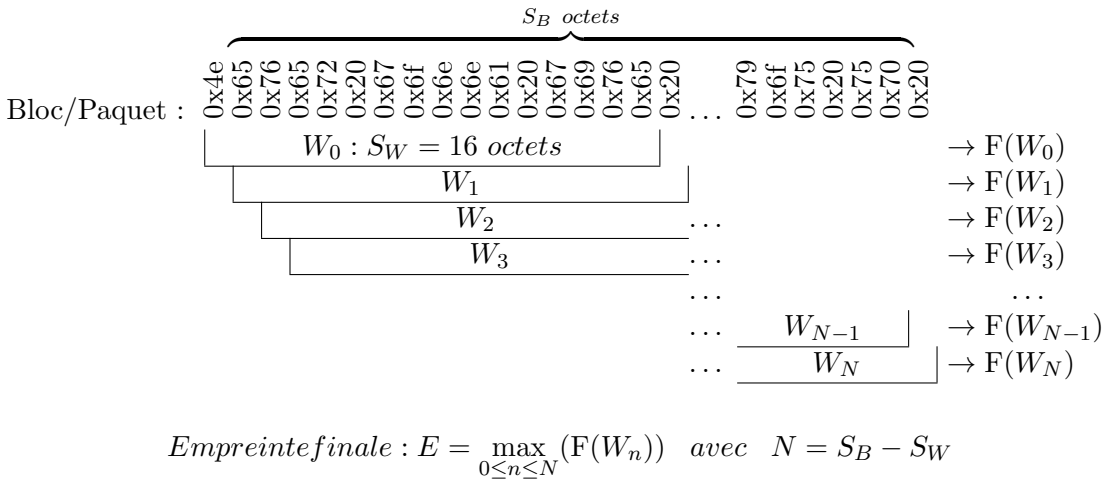

Figure 2.10 Illustration de l'algorithme de *max-Hashing*

Dans le cas du référencement cette empreinte pourra être sauvegardée dans une base de données et dans le cas de la détection cette empreinte sera comparée à celles contenues dans la base de données. David [11] suggère l'utilisation d'une table de hachage comme base de données, l'empreinte calculée servant d'index dans cette table. Cela procure un avantage très intéressant : une fois la taille de la base de données fixée, rajouter de nouvelles signatures à détecter ne ralentit pas directement le processus de détection contrairement aux expressions rationnelles. Si l'empreinte est bel et bien retrouvée, on peut considérer qu'il est très probable que le paquet contienne un fragment de fichier connu.

2.4 Matériels utilisés dans le domaine d'analyse des flux réseaux

La vitesse des réseaux et la capacité de stockage de nos supports augmentant sans cesse, il devient primordial de proposer des implémentations qui permettent de supporter ces évolutions. Or, les solutions logicielles sur CPU montrent des limitations dans les domaines de la détection de données connues. La tendance actuelle consiste donc à déporter les traitements lourds sur du matériel spécialisé.

Dans ces domaines, on présente généralement quatre types de matériel spécialisé : les processeurs réseau, les ASIC, les FPGA et les GPU. Une comparaison entre ces équipements et les CPU a été réalisée par Smith *et al.* [54] dans laquelle ils présentent les GPU comme une plateforme particulièrement appropriée pour le calcul de signatures sur des paquets réseau. Des observations similaires ont été faites par Vasiliadis *et al.* pour l'adaptation de l'IDS `Snort` [48] sur GPU, `Gnort` [60], et pour celle de l'antivirus `ClamAV` [10], `GrAVity` [61], avec

des vitesses de traitement entre 2× et 100× supérieures (comparées aux versions purement CPU). Le traitement des expressions régulières profite aussi de grandes accélérations sur ces matériels. Des algorithmes inspirés de la méthode d'Aho et Corasick [2] et implémentés sur GPU permettent, par exemple, de traiter jusqu'à 143 Gb/s [8] et sont 15× plus performantes que sur CPU [37]. Enfin, le domaine judiciaire n'est pas en reste, puisque ses outils sont eux aussi transposés sur GPU afin de profiter de son architecture [33].

Il est à noter que l'algorithme de *max-Hashing* a été implémenté sur FPGA par David [11] et sur GPU par Lerebours [29], les deux implémentations ne permettant pas de supporter des débits supérieurs à 10 Gb/s. De plus, le processus de référencement mentionné dans ces documents est générique pour tous les types de fichiers et ne traitent, pour la détection, que du DPI (*Deep Packet Inspection*, analyse des champs de données). En revanche, il y est fait mention de la nécessité d'extraire, lors du référencement, des empreintes originales depuis les segments qui sont propres au fichier.

2.5 Algorithmes de référencement et de détection dédiés aux fichiers vidéo

Plusieurs méthodes existent spécifiquement pour le référencement et la détection de fichiers multimédias connus. Mentionnons, notamment, le tatouage numérique [1] et la détection par le contenu [30]. Le tatouage numérique consiste en l'insertion d'informations supplémentaires dans la ou les images du fichier. Si ces informations sont visibles à l'œil humain alors cela s'apparente à l'ajout d'un filigrane. En revanche si ces informations doivent rester invisibles on parle alors de stéganographie. La détection des images marquées consiste donc à retrouver les informations qui y ont été ajoutées. La détection par le contenu consiste, quant à elle, à référencer puis détecter des informations visuelles natives du fichier comme par exemple la chrominance, la luminance, l'histogramme d'une image ou d'une partie de celle-ci etc. Il existe de nombreux algorithmes pour ces deux approches [1, 30]. Ils ont, en général, l'avantage d'être très permissifs aux modifications des images originales : le contenu d'un fichier peut, dans une certaine mesure, toujours être détecté malgré les ré-encodages, les retouches, les insertions et suppressions d'images par exemple [4, 42]. L'algorithme de *max-Hashing* étant, lui, concentré sur la représentation binaire des fichiers, de telles modifications visuelles affecteraient considérablement notre capacité à reconnaitre les fichiers originaux référencés. En revanche, le tatouage numérique et la détection par le contenu requièrent la reconstitution d'au moins une partie du flux vidéo afin de pouvoir identifier si les images sont connues ou non. On le rappelle, cette reconstitution est délicate, voire impossible, lorsque l'information est disséminée dans un flux de paquets non-ordonnés, incomplets et provenant de nombreux utilisateurs.

2.6 Conclusion du chapitre

Dans ce chapitre, nous avons présenté différents algorithmes permettant la détection de fichiers connus ainsi que les matériels exploités pour les implémenter. Nous avons identifié que les méthodes de hachage sont préférables aux méthodes de comparaison de chaînes de caractères dans le cadre de notre problématique, c.à.d la détection de millions de fichiers connus illégaux. Parmi les algorithmes de hachage, le *max-Hashing* semble le plus adapté à la détection de segments de fichiers connus disséminés dans des paquets réseau de taille inconnue. Néanmoins, comme indiqué par David [11] et Lerebours [29] et comme nous aurons l'occasion de le montrer, il convient de proposer une méthode de référencement qui génère des empreintes originales, qui ne se retrouvent pas dans d'autres fichiers. Enfin, nous présenterons la poursuite des travaux de Lerebours sur l'implémentation de l'algorithme de *max-Hashing* sur GPU pour la détection de segments de fichiers connus et de flux TCP/UDP-IP connus pour des débits de 40 Gb/s et plus.

CHAPITRE 3

RÉFÉRENCEMENT DE FICHIERS VIDÉO

Le référencement consiste en la génération des empreintes de référence. Celles-ci pourront être, par la suite, comparées aux empreintes qui seront générées sur les paquets réseau. Une empreinte est dite *de qualité*, ou *originale*, lorsque celle-ci ne peut être générée que depuis un segment de données très particulier. La qualité des empreintes a donc un impact direct sur les performances de la détection en réduisant notamment la probabilité d'apparition de faux-positifs. Il est, par ailleurs, important de souligner que le processus de référencement n'est pas particulièrement restreint dans son temps d'exécution : on peut se permettre de complexifier cette tâche si cela permet de générer des empreintes de meilleure qualité.

Dans ce chapitre, nous discuterons dans un premier temps de la notion d'"entropie" dans un fichier. On dira d'un segment de données qu'il est original ou à "haute-entropie" lorsqu'il est très peu probable de retrouver ce segment dans d'autres fichiers. A l'inverse, on dira d'un segment qu'il est peu original, commun ou à "faible-entropie" lorsqu'il est très probable de le retrouver dans plusieurs fichiers. Puis nous nous concentrerons sur un type de fichier en particulier : les fichiers vidéo sous format MP4 et utilisant l'encodeur H.264. Cette étude de l'"entropie" dans les fichiers vidéo nous permettra de concevoir une méthode de référencement dédiée à ces fichiers. Enfin, nous présenterons les performances de cette nouvelle méthode en les comparant à celles de la méthode générale.

3.1 Originalité des segments de données dans un fichier

La qualité d'une empreinte reflète l'originalité du segment d'information depuis lequel elle a été générée. Si ce segment se retrouve dans d'autres fichiers, alors la même empreinte en sera probablement extraite : on peut donc dire qu'il ne s'agit pas d'un segment *propre* au fichier. Étant donné que ce segment, seul, ne nous donne pas suffisamment d'information pour pouvoir déterminer d'où il est extrait, on dira alors que son "entropie" est faible. Illustrons ce concept d'originalité pour différents types de fichiers et d'informations.

3.1.1 Fichiers textuels, analogie avec le langage

A travers l'histoire, chaque mot de notre langue a déjà été employé. Cependant, en assemblant plusieurs mots, on peut constituer une phrase qui, elle, n'aura peut être jamais été

formulée auparavant. Ainsi, si l'on devait rechercher dans un corpus de textes le poème **Le Corbeau** de Edgar Allan Poe (traduit par Charles Baudelaire) il serait plus judicieux de vérifier les occurrences de la phrase « Que cette parole soit le signal de notre séparation, oiseau ou démon ! » plutôt que celles du mot « oiseau » que nous retrouverons très certainement dans d'autres textes.

Néanmoins, comme nous l'avons vu dans le chapitre précédent, se baser sur des fragments d'information trop longs peut poser plusieurs problèmes, la probabilité que ceux-ci se retrouvent segmentés ou altérés augmentant avec leur taille. Idéalement, il faudrait sélectionner une fenêtre plus petite mais tout aussi unique : un nombre relativement faible de caractères contigus qu'il serait peu probable de retrouver dans d'autre textes. Dans ce cadre précis, notre connaissance de la langue française, et de son utilisation, nous permettent d'avancer que la chaine de caractères "sur le minuit lugubre" formerait une fenêtre originale.

3.1.2 Fichiers HTML, le cas des balises

Le langage HTML est utilisé pour structurer les pages web. Cette structure consiste principalement en l'insertion de balises et de méta-données afin de délimiter chaque section de la page, comme les titres ou les paragraphes. On présente un exemple de fichier HTML élémentaire à la figure 3.1. On repère les dites balises grâce aux caractères "<" et ">". Dans ce cadre là, une connaissance accrue du langage HTML et des spécifications délivrées par le W3C [63] nous permet de localiser et de filtrer les fenêtres qui ne seraient pas propres à ce fichier là en particulier. Typiquement, les balises de l'exemple peuvent se retrouver dans des millions d'autres fichiers : ce sont des secteurs à "faible-entropie" qu'il vaudrait mieux éviter si l'on souhaite obtenir des empreintes *originales*.

```
<!DOCTYPE html>
<html lang="fr">
  <head>
      <meta charset="utf-8">
      <title>Bonjour Monde !</title>
  </head>
  <body>
      <h1>Un gros titre !</h1>
      <p>Voici un premier paragraphe.</p>
  </body>
</html>
```

Figure 3.1 Exemple de code HTML5

3.1.3 Formats de fichiers et encapsulation

On peut généraliser les observations précédentes à tous les formats de fichiers. Ces derniers possèdent, pour la quasi-totalité, des parties distinctes structurées et délimitées par des balises, des entêtes (*headers*) et des fins de sections/fichiers (*footers*). Ces champs étant génériques on les retrouvera dans plusieurs fichiers de même format. Un problème persiste encore pour les données inclues entre ces balises : pouvons-nous considérer que celles-ci sont propres au fichier dans lequel elles sont stockées ?

Roussev et Garfinkel [52] répondent en partie à cette question avec l'exemple du format PDF. Ce format, notamment utilisé pour la description de pages, contient typiquement des données de texte et d'images dans ses descriptions. Mais ces données sont, de fait, compressées en utilisant des formats tels que zlib ou deflate concernant le texte, et JPEG ou JPEG2000 pour les images. Par conséquent, ces champs de données vont eux aussi être structurés et probablement inclure des zones à "faible-entropie". Nous nous attardons sur un type de format dont la fonction première est de contenir des informations dans d'autres formats : un conteneur.

3.2 Fichiers vidéo : étude du format MP4 avec H.264

Un fichier vidéo est, comme son nom l'indique, un fichier informatique qui contient des données représentant des informations visuelles, auditives et éventuellement textuelles (sous-titres, annotations). Ces informations sont généralement encodées et compressées. Elles nécessitent donc un traitement avant de pouvoir être présentées. On entend par "présentation" le fait d'afficher une succession d'images, pour l'information visuelle, ou d'émettre un son, pour l'information auditive. Un fichier vidéo est donc caractérisé par sa présentation : les images et les sons qu'il permet de générer. Si cette présentation est unique (différente de celles générées par d'autres fichiers) alors on peut considérer qu'elle est *propre* au fichier qui la contient. Par extension, les informations encodées et compressées, qui ont permis de générer cette présentation, seraient donc elles aussi *propres* au fichier dans lesquelles elles sont stockées.

MP4, ou MPEG-4 *part* 14, est un format conteneur multimédia qui peut inclure des flux vidéo(s) et audio(s) ainsi que des sous-titres et des méta-données. Il étend les spécifications du standard MPEG-4 *part* 12, lui-même basé sur le format de fichier *QuickTime*. Les informations audio et visuelles peuvent être encodées dans les différents formats compatibles avec MP4, parmi lesquels on notera H.264. Aussi appelé MPEG-4 Part 10 ou MPEG-4 AVC (*Advanced Video Coding*), il s'agit du format d'encodage de l'information visuelle dédié à

la vidéo le plus utilisé à l'heure actuelle. Par la suite, nous allons détailler les structures de ce conteneur et de cet encodeur afin de déterminer l'emplacement des segments à "haute-entropie" dans un fichier MP4 avec H.264. Nous nous baserons sur les documents des normes correspondantes [55, 56].

3.2.1 Conteneur vidéo MP4

Dans un fichier MP4, les informations sont divisées dans des blocs aussi appelés *atomes*. Un atome est généralement constitué d'un champ indiquant sa taille, d'un champ d'entête et d'un champ de données qui peut lui aussi contenir d'autres atomes. C'est la succession d'*atomes* et leur encapsulation qui forment la structure, le squelette, du fichier MP4. La structure minimale d'un fichier MP4 est présentée à la figure 3.2. Les champs `ftyp`, `moov`, `trak` et `mdat` sont des atomes et ont chacun une fonction bien spécifique :

`ftyp`
: Les données inclues dans cet atome permettent de spécifier le type et la version du fichier présentement manipulé. Ces données doivent donc être compatibles avec les différentes spécifications impliquées par le type (structure, encodage etc.). C'est ici que l'on pourra signaler qu'il s'agit bien d'un fichier MP4.

`moov`
: Ce champ est le conteneur pour de nombreuses méta-données concernant le fichier et les flux (vidéos, audios ou textuels) qui y sont inclus.

`trak`
: Cet atome doit nécessairement être inclus dans l'atome `moov`. Il contient les méta-données concernant un flux en particulier. On y indiquera, par exemple, le nombre d'échantillons et leur encodage.

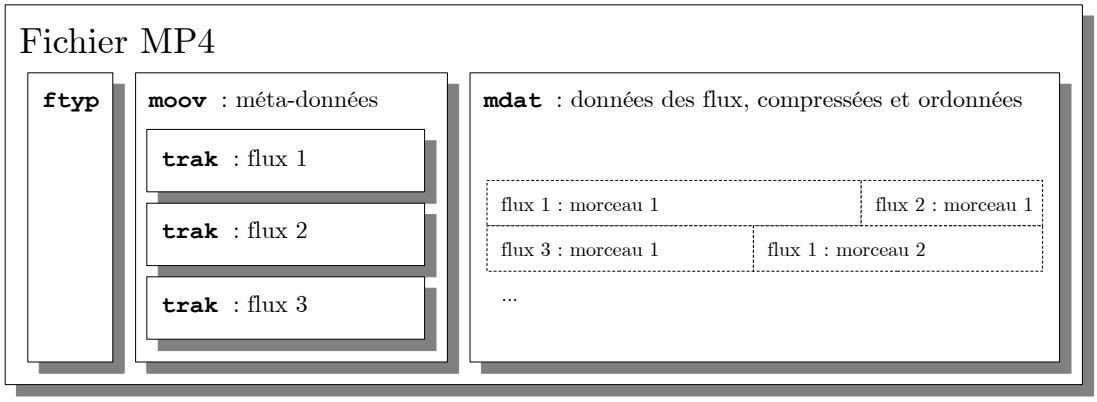

Figure 3.2 Structure d'un fichier MP4 élémentaire

`mdat`

C'est dans cet atome que l'on trouvera les données encodées et compressées relatives aux différents flux. Les données sont agrégées en "morceaux" (*chunks* en anglais) qui regroupent un nombre variable d'échantillons. Les morceaux sont ordonnées pour faciliter la décompression et la présentation de ceux-ci.

À la lecture des précédentes descriptions, on peut déterminer que les informations contenues dans les atomes `ftyp`, `moov` et `trak` ne sont pas, a priori, propres au fichier dans lequel on les trouverait. Les fichiers de même type et ayant des flux codés de manière similaire ne seraient pas clairement différentiables à partir des informations contenues dans ces champs-là. En revanche, les données contenues dans l'atome `mdat` semblent être de meilleures candidates pour représenter le fichier. Étudions le cas de données contenues dans un tel atome et utilisant le codeur H.264.

3.2.2 Codage et compression vidéo avec H.264

Avant d'observer l'organisation de l'information avec H.264, il convient de présenter sommairement le fonctionnement de cet encodeur afin d'en mieux comprendre la structure. La compression d'un flux vidéo se base sur les fortes redondances observables dans une même image (un motif qui se répète, par exemple) et d'une image à l'autre dans le flux. Deux images consécutives sont, en général, très similaires : les tintes changent peu, voire pas du tout, et les éléments restent immobiles ou se déplacent à l'intérieur du cadre. On peut donc utiliser ces répétitions pour réduire la taille des informations à transmettre/enregistrer.

Avec H.264, on divise l'image en petites portions ou blocs aussi appelés *Macro-Blocs* (MB). Ils comportent les informations visuelles de chrominance et de luminance compressées (par l'application de la transformée en cosinus discrète puis quantification vectorielle) et, éventuellement, une information de déplacement. Une image déjà reconstruite peut être utilisée pour la construction d'autres images : les informations contenues dans les MB indiquent alors les changements de tinte et les déplacements par rapport à l'image de référence. L'information visuelle, propre au fichier, est donc principalement contenue dans les données portées par les *Macro-Blocs*. Ces derniers feraient donc des fenêtres idéales pour le calcul d'empreintes originales.

L'encapsulation des données des *Macro-Blocs*, avec H.264, est réalisée de la manière suivante (voir la Figure 3.3 pour une illustration visuelle de cette encapsulation) :

– Les données des *Macro-Blocs* (variations de chrominance/luminance et vecteurs de déplacement) sont accompagnées d'un entête permettant de spécifier la configuration du MB et, éventuellement, l'image de référence à laquelle il se rapporte.
– Les MB sont ensuite groupés en tranches (ou *slices*) elles-mêmes dotées d'un entête. Toutes les données d'une tranche peuvent ensuite être compressées en utilisant un codeur entropique de type CABAC, CAVLC ou exponentiel-Golomb [32, 46].
– Après l'étape de compression, il est probable que la tranche occupe un nombre de bits non-multiple de 8. Afin de maintenir un alignement au niveau de l'octet, on rajoute des bits nuls (0) à la fin de la tranche pour compléter un octet entamé. Le tout forme alors ce qui est appelé une *Raw Byte Sequence Payload* (RBSP).
– La dernière encapsulation forme une *Network Abstraction Layer (NAL) unit* à partir du RBSP et d'un entête. Cet entête comporte notamment le type de *NAL unit*, et un grand nombre de drapeaux (*flags*) indiquant la configuration des informations inclues et des instructions pour un décodage approprié de celles-ci.

3.3 Utilisation du logiciel `FFmpeg` pour le calcul des signatures

`FFmpeg` [15] est un logiciel de traitement de flux multimédias. Il est sous licence libre, et ses fichiers sources, écrits en langage C, sont ouverts. De plus, il se base sur les bibliothèques `Libav` [31] intégrant de nombreux codeurs/décodeurs ("codecs"). Nous avons choisi d'utiliser ce logiciel et ces bibliothèques afin, d'une part, de limiter le temps de développement et, d'autre part, pour profiter de la robustesse et de la polyvalence reconnues de celles-ci.

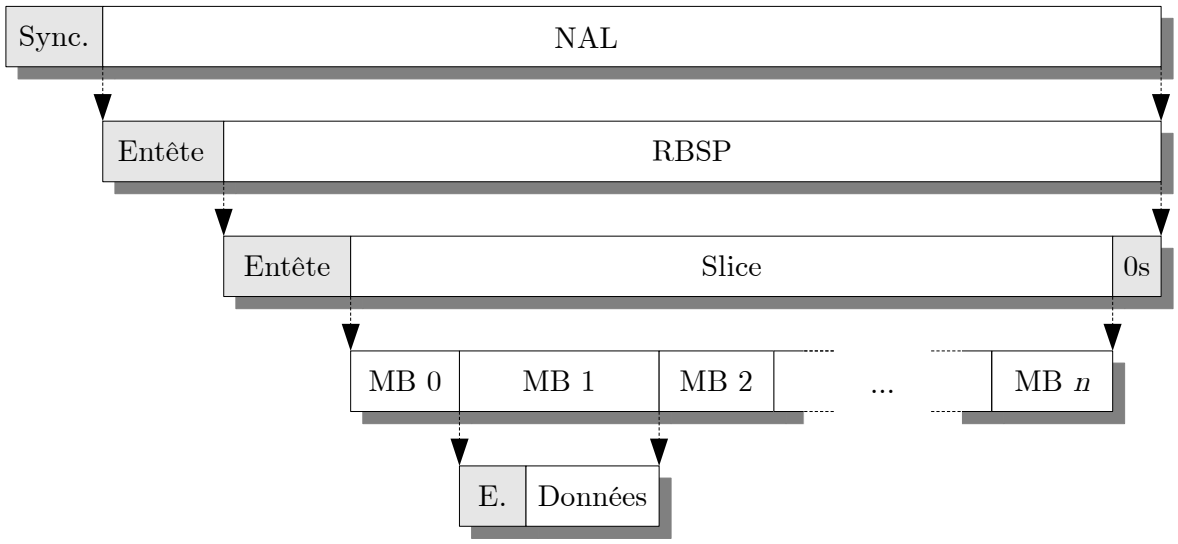

Figure 3.3 Encapsulation des données avec H.264

3.3.1 Modification des fichiers sources de `FFmpeg`

Nous souhaiterions donc filtrer les zones à "faible-entropie" du fichier pour nous concentrer sur les données inclues au niveau *Macro-Bloc* de H.264. Le recours aux fonctions inclues dans les bibliothèques de `FFmpeg` nous permet de filtrer les champs du conteneur (MP4) ainsi que les méta-données n'influant pas sur la reconstruction des images. Les fonctions utilisées, dont les prototypes sont présentées à la figure 3.4, donnent en effet directement accès aux données nécessaires au décodage de chaque image. Par la suite, on se concentre sur le code source du décodage H.264. Sa lecture et sa compréhension sont grandement facilitées par la lecture des recommandations de la *International Telecommunication Union* (ITU-T) pour H.264 [22]. La position, la taille et la fonction de chaque champ, chaque donnée et chaque drapeau (*flag*) y sont clairement référencés. Dans le code source de `FFmpeg`, nous pouvons donc suivre le traitement de tous ces champs jusqu'aux segments que nous avons préalablement choisis : les données des *Macro-Blocs*.

On ajoute de nouvelles lignes dans le code original afin de pouvoir enregistrer, au fil du décodage du fichier par les bibliothèques, les informations permettant de déduire la position des données des *Macro-Blocs*. Ces informations sont enregistrées dans un espace mémoire tampon qui sera lu une fois le décodage terminé. On ajoute au code C de `FFmpeg`, tout d'abord, les ressources permettant l'échange de données entre la bibliothèque, qui écrira dans la mémoire, et un autre programme lié à cette bibliothèque qui lira et compilera toutes les informations contenues dans la mémoire tampon. On peut lire, à la Figure 3.5, une partie du code ajouté au code source original de `FFmpeg` dans lequel on déclare les variables enregistrant l'emplacement de l'espace de mémoire tampon et indiquant la position du premier octet libre dans celui-ci (un index). On y joint aussi des fonctions permettant au programme extérieur à ces bibliothèques de modifier et lire ces variables. Les autres ajouts de code, non reportés dans ce document, réalisent l'enregistrement dans la mémoire tampon des informations de décalage (*offset*) et de position des *Macro-Blocs*. Enfin, le programme utilisant cette bibliothèque récupère toutes les informations inscrites dans la mémoire, et les compile pour être utilisables lors du référencement : pour chaque zone à "haute-entropie" on indique la position de son premier octet dans le fichier, la taille de la zone et l'horodatage (*timestamp* en anglais) de l'image du flux vidéo concernée par ce segment de données.

```
/* Ouverture du fichier */
int avformat_open_input(AVFormatContext ** ps,
                        const char * fichier,
                        AVInputFormat * fmt,
                        AVDictionary ** options);

/* Obtention des informations sur les flux */
int avformat_find_stream_info(AVFormatContext * ic,
                              AVDictionary ** options);

/* Initialisation d'un codec */
int avcodec_open2(  AVCodecContext * avctx,
                    const AVCodec * codec,
                    AVDictionary ** options);

/* Copie des donnees d'une image */
int av_read_frame(AVFormatContext * s,
                  AVPacket * pkt);

/* Decodage des donnees */
int avcodec_decode_video2(AVCodecContext * avctx,
                          AVFrame * image,
                          int * got_image_ptr,
                          const AVPacket * avpkt);
```

Figure 3.4 Prototypes des fonctions de FFmpeg (Libav) utilisées

```
/* Pointeur vers memoire tampon et index dans ce tampon */
void * ExtractorBuffer = NULL;
long long ExtractorIndex = 0;

/* Modification de la valeur de l'index */
void attribute_align_arg avcodec_setExtractorIndex(long long val){
  ExtractorIndex = val;
  return;
  }

/* Lecture de la valeur de l'index */
long long attribute_align_arg avcodec_getExtractorIndex(void){
  return ExtractorIndex;
  }

/* Modification de la valeur du pointeur */
void attribute_align_arg avcodec_setExtractorBuffer(void * p){
  ExtractorBuffer = p;
  return;
  }
```

Figure 3.5 Ressources ajoutées à la bibliothèque de FFmpeg

3.3.2 Calcul des signatures

Une fois les zones à “haute-entropie” localisées dans le fichier vidéo, le processus de calcul de signature diffère très peu de la méthode “brute”. Le fichier doit donc être divisé en blocs et les empreintes ayant la valeur maximale dans chaque bloc sont retenues. On ne conserve que les empreintes qui ont été calculées à partir des zones à “haute-entropie” déterminées précédemment, les autres empreintes ne sont pas conservées : elles ne sont plus probablement communes à d'autres fichiers. L'approche consistant à calculer les empreintes directement sur les zones à “haute-entropie” en négligeant les autres segments, bien que simple, fut tout de suite éliminée. En effet, dans un tel cas il est probable que des empreintes de valeurs supérieures puissent être calculées dans le voisinage de nos zones à “haute-entropie”. Lors de la détection, ces empreintes occulteraient celles que nous aurions sauvegardé. La taux de faux-négatif (ne pas reconnaitre un segment pourtant référencé) augmenterait alors significativement.

3.4 Méthodologie pour la mesure des redondances et des faux-positifs

Nous souhaitons à présent mesurer et comparer la qualité des empreintes générées depuis des fichiers vidéo lors des référencements “brut” et amélioré. Dans ce but, nous utiliserons la banque de vidéos CCV [23] qui contient 8073 fichiers MP4 uniques. Elle compte 182,4 heures d'informations vidéo et audio variées. Nous procéderons à deux tests pour chacun des référencements.

Le référencement d'un fichier n'est, en général, réalisé qu'une seule fois. Les empreintes qui représentent ce fichier et leur nombre sont donc déterminées lors de ce processus. Or, si une empreinte est commune à au moins deux fichiers (on parle alors de redondance) et qu'on ne peut pas, techniquement, enregistrer et différencier ses deux occurrences alors elles ne pourront pas être prises en compte lors de la détection. Les fichiers “perdraient” donc une empreinte pour les représenter. Dans le cas de figure où ces redondances seraient fréquentes, un fichier verrait son capital d'empreintes s'amoindrir au fur et à mesure que de nouveaux fichiers seraient ajoutés à la base de données de références. Cela réduirait donc la probabilité de détecter les fichiers référencés (faux-négatifs). Notre premier test permettra de mesurer les apparitions des redondances en comparant toutes les empreintes calculées lors du référencement des fichiers de la base de données.

Lors du deuxième test, on sauvegardera les empreintes calculées sur une moitié de la base de données de fichiers vidéo. Ces empreintes serviront lors de l'analyse du transfert de la

deuxième moitié de la BDD sur un lien Ethernet. Nous utilisons ici le système de détection complet dont les caractéristiques et les performances sont mesurées au chapitre 4. Il est à noter que dans le cadre de ce test, nous garderons les empreintes redondantes afin de mesurer le taux de faux-positifs que celles-ci engendreraient.

3.4.1 Méthode de référencement "brute"

Le référencement "brut" consiste à diviser les fichiers en blocs, à les hacher, et à en extraire les 4 maxima locaux (pour les 4 combinaisons des deux bits de poids fort "00", "01", "10" et "11"). Pour chaque fichier nous fixons le nombre de blocs à 512 soit $512 \times 4 = 2048$ empreintes générées par fichier. Les résultats sont présentés au Tableau 3.1 et sont détaillés ci-dessous.

Redondance et originalité des signatures

Sur la totalité de la base de données CCV, nous calculons $4 \times 512 \times 8073 = 16.533.504$ empreintes. En les comparant les unes aux autres, nous déterminons que 243 fichiers génèrent 371 empreintes redondantes (communes à au moins deux fichiers). Une analyse approfondie de la position de ces empreintes dans les fichiers indique tout d'abord que 357 d'entre elles sont calculées sur les données audio communes à 7 fichiers qui possèdent la même bande sonore. D'autre part, nous observons que les 14 empreintes restantes sont hautement redondantes puisqu'elles sont communes à près de 143 fichiers. Celles-ci sont calculées dans l'atome `moov` (voir section 3.2.1), sur les méta-données du flux vidéo.

Détection et taux de faux-positifs observés

Lors du transfert de la deuxième moitié de la base de données, près de 500 millions d'empreintes sont calculées sur les paquets Ethernet. Parmi celles-ci, 18000 empreintes sont signalées comme ayant été référencées (au moins une fois). Étant donné qu'aucun des fichiers référencés n'est analysé ici, il s'agit donc de **18000 faux-positifs**. Cela équivaut à un taux de faux-positifs observés de l'ordre de $\frac{3,8}{100.000}$. Il convient de garder à l'esprit que lorsqu'une empreinte générée depuis les paquets Ethernet est commune à plus d'un fichier référencé, celle-ci ne génère qu'un seul faux-positif.

Tableau 3.1 Résultats des deux méthodes de référencement

	Redondance(s)	Faux-positif(s)
Méthode "brute" :	357	18000
Méthode améliorée :	*0*	*0*

3.4.2 Méthode améliorée du référencement

La méthode améliorée consiste à ne conserver que les maxima qui ont été calculés sur des zones à “haute-entropie”. Comme précédemment, nous conserverons 4 empreintes sur les 512 blocs de chaque fichier.

Redondance et originalité des signatures

Les résultats pour l'algorithme brut ont montré que les redondances étaient principalement calculées depuis les données audio et les méta-données. Or, notre méthode améliorée fut conçue pour précisément éviter ces champs-là. Avec notre nouvelle méthode, nous ne retrouvons aucune redondance lors du référencement des 8073 fichiers : toutes les empreintes calculées semblent être propres à leurs fichier d'origine. Il s'agit en tout cas de nos observations dans le cadre de notre base de données. Il pourrait être intéressant de constituer une base de données de fichiers vidéo plus importantes afin de confirmer nos observations.

Détection et taux de faux-positifs observés

Nous n'observons aucun faux-positif lors de ce test : aucune des 500 millions d'empreintes n'est commune à celles référencées avec notre méthode dédiée aux fichiers vidéo et à l'extraction de leurs zones à “haute-entropie”. Ici encore, l'utilisation d'une banque de vidéos plus importante permettrait de mesurer les limites de notre méthode.

3.5 Discussion

On pourra remarquer que le processus d'analyse des fichiers est relativement lent puisque le code de `FFmpeg` qui a été modifié réalise plus qu'une recherche des données à “haute-entropie” : il décode le fichier afin de générer le flux d'images. Néanmoins, comme mentionné auparavant, l'étape de référencement des fichiers n'a pas de contrainte de temps à respecter. Cette lenteur est l'inconvénient principal de l'utilisation des bibliothèques de `FFmpeg` qui ont été, en revanche, très simples à modifier et à exploiter.

3.6 Conclusion du chapitre

Au cours de ce chapitre, nous avons d'abord présenté la notion d'“entropie” dans un document puis, on a montré qu'une approche simple du référencement avec l'algorithme de *max-Hashing*, sans prise en compte de l'“entropie”, ne permettait pas de générer des empreintes originales. Nous constatons en effet un nombre conséquent de redondances et de faux-positifs avec cette approche native du référencement. Il a donc été proposé de s'intéresser

de plus près à la structure des fichiers référencés (MP4 avec H.264) afin de localiser les zones à “haute-entropie” qui permettraient de générer des empreintes originales. Les positions de ces zones à “haute-entropie”, confinées dans les données des *Macro-Blocs*, sont extraites grâce aux bibliothèques modifiées de `FFmpeg`. Le référencement amélioré consiste alors à ne conserver que les empreintes maximales calculées sur les zones à “haute-entropie”. Avec ces nouvelles empreintes, et sur nos 8073 fichiers vidéo, nous ne constatons plus aucune redondance ni aucun faux-positif. On dispose donc d’empreintes de qualité qui pourront être intégrées à notre système de détection, détaillé dans le chapitre suivant.

CHAPITRE 4

SYSTÈME DE DÉTECTION

Nous pouvons à présent nous consacrer au système de détection de segments de fichiers connus (référencés) dans des paquets réseau. Pour le système, nous disposons d'une interface réseau 40 GbE de Mellanox, de deux CPU Intel Xeon E5-2609 accompagnés de 16 Go de mémoire centrale (DDR3) et d'une carte GPU NVIDIA K20c. Sachant que nous souhaitons déléguer les tâches d'analyse au GPU, on peut anticiper que de multiples copies seront nécessaires pour apporter les paquets reçus à la carte graphique. La figure 4.1 présente l'architecture globale du système, ainsi que les interconnexions et une partie des copies nécessaires pour transférer un paquet de l'interface réseau au GPU.

Toutes les tâches du système seront passées en revue avant de présenter leurs performances. Nous commencerons par traiter celles en amont de toutes les autres : le réception des paquets Ethernet et leur copie vers la mémoire embarquée du GPU. Enfin, une architecture complète du système sera présentée suivie des résultats de nos tests sur celle-ci.

4.1 Réception et copies des paquets Ethernet

4.1.1 Interface réseau et CPU

L'interface réseau reçoit les paquets au travers des ports physiques qu'elle possède. Après avoir vérifié leur intégrité, un signal (interruption) est envoyé au système d'exploitation (SE) indiquant que des paquets sont prêts à être transférés au CPU (mémoire centrale). Le SE stockera ces paquets une première fois dans un espace réservé (non-accessible par les applications), puis les copiera un seconde fois pour les applications qui en font la demande. Il

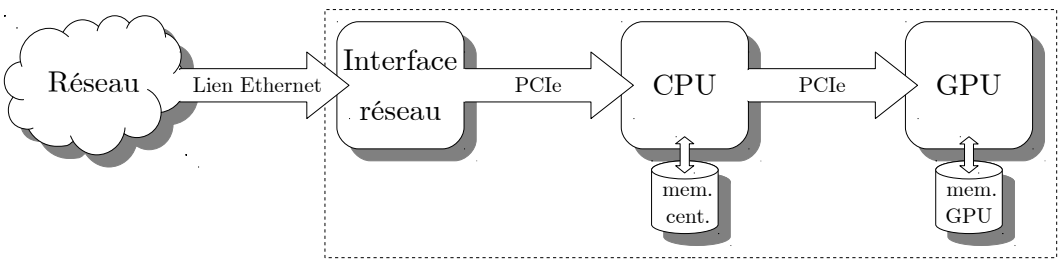

Figure 4.1 Copies des paquets au travers du système

s'agit d'un des problèmes mentionnés par Lerebours [29] : les multiples copies réalisées par le système d'exploitation dans la mémoire centrale de l'ordinateur ainsi que les temps de traitement des interruptions ne permettraient pas de supporter des débits élevés (10 Gb/s et plus).

Notre objectif consistant à traiter des débits pouvant atteindre 40 Gb/s, nous avons tenté de traiter ce problème avec l'utilisation de `PF_RING`. Développé par NTOP [39], `PF_RING` est un module qui se substitue au système d'exploitation dans le processus de réception des paquets et de leur mise à disposition auprès des applications. Les multiples copies précédemment évoquées sont évitées en autorisant les applications à accéder aux paquets le plus tôt possible. Un autre avantage de `PF_RING` est son support de la bibliothèque `pcap` [57] : il n'est pas nécessaire de modifier un code utilisant `pcap` pour qu'il puisse profiter de `PF_RING`.

4.1.2 Agrégation des paquets puis envoi vers le GPU

L'intérêt de l'utilisation du GPU réside dans son architecture hautement parallèle qui permet de réaliser le même traitement sur des données différentes simultanément. Afin de tirer parti de sa puissance pour le traitement de paquets, il convient par conséquent de lui fournir une quantité de données suffisante. Cela permet, d'autre part, de réduire le surcoût des transactions à effectuer via le bus PCIe. On note que les transferts sont nativement réalisés en DMA (*Direct Memory Access*) par les pilotes fournis par NVIDIA.

Stratégie de stockage

Chaque fois qu'un paquet est reçu, une fonction dédiée de notre application est appelée (fonction de *callback*). Celle-ci aura pour rôle de stocker temporairement le paquet dans un espace mémoire tampon aux côtés d'autres paquets. Une fois cette espace plein, ou après un temps donné, il est intégralement copié sur la mémoire embarquée de la carte GPU afin d'en traiter le contenu. Ce traitement est massivement parallèle : les fils d'exécution (*threads* en anglais) opèreront en parallèle sur les données qui leur seront attribuées. Sachant que la taille des paquets Ethernet reçus est variable (entre 64 et 1518 octets), certains fils auraient un travail bien plus court que d'autres (rapport de 1 à 24). Cela pourrait potentiellement se solder par une sous-utilisation des ressources sur GPU et donc un ralentissement global de l'application.

Pour contourner ce problème, on choisit de confier à chaque fil une part de taille fixe d'un paquet. Lors du stockage temporaire, avant le transfert vers le GPU, on prendra donc soin d'aligner le début de chaque paquet sur une adresse multiple de la taille des parts. Des

octets nuls (0) seront insérés entre la fin de chaque paquet et le début du suivant en mémoire (zone de bourrage). Ainsi, au prix du transfert d'octets "inutiles", on s'assure d'une utilisation optimale des ressources du GPU.

Mémoire non-paginée

Usuellement, les ressources en mémoire utilisées par les programmes sont paginées. Cela signifie que ces espaces sont divisés en pages qui peuvent être déplacées en dehors de la mémoire centrale par le CPU. Elles pourront par la suite être ramenées lorsqu'elles sont de nouveau requises. Cela permet aux programmes de demander des espaces de mémoire éventuellement plus larges que la mémoire centrale elle-même. En revanche, lorsqu'une page a quitté la mémoire centrale, le temps pour y accéder augmente drastiquement. Dans le cas de notre application, stocker temporairement les paquets reçus dans une mémoire tampon non-paginée permettrait donc d'obtenir de meilleurs résultats lors des transferts réalisés par DMA. Les bibliothèques CUDA fournies par NVIDIA proposent une fonction (figure 4.2) pour l'allocation de telles mémoires dans le cadre de transferts vers le GPU. Elles permettent, par ailleurs, de profiter des fonctionnalités asynchrones de celui-ci (voir section 4.3.5). En revanche, il faut veiller à laisser suffisamment de mémoire centrale libre pour le bon fonctionnement du système d'exploitation et des autres programmes.

4.2 Processeurs graphiques (GPU) et programmation

Les processeurs graphiques, comme leur nom l'indique, étaient à l'origine purement dédiés aux calculs graphiques en trois dimensions. Il fut très rapidement remarqué que leur architecture était tout aussi efficace pour d'autres types de calculs : ainsi naquît le principe de GPGPU (*General-purpose Processing on Graphics Processing Unit*). Ce "détournement" de leur usage original est aujourd'hui si répandu que les constructeurs conçoivent des gammes de GPU purement dédiées pour ces nouvelles utilisations. Les GPU sont des plateformes privilégiées pour le calcul de signatures sur des paquets réseau comme nous avons pu le montrer à la section 2.4. On les préfère aux CPU notamment grâce à leurs performances (voir tableau 4.1). Faces aux FPGA, c'est à la fois le coût et la facilité de programmation qui profitent aux GPU.

```
1 /* Allocation de memoire */
2 cudaError_t cudaHostAlloc(void ** ptrHote,
3                            size_t taille,
4                            unsigned int drapeau); //cudaHostAllocPortable
```

Figure 4.2 Prototype de la fonction d'allocation de mémoire de CUDA

Les processeurs graphiques sont en général déportés sur une carte périphérique, dotée de sa propre mémoire DRAM et connectée au reste du système via un bus PCIe. Dans cette section, nous présenterons les spécificités de l'architecture CUDA pour les processeurs graphiques ainsi que la programmation logicielle associée. Nous utiliserons autant que possible la nomenclature anglaise utilisée par NVIDIA [40].

4.2.1 Architecture du processeur graphique Kepler GK110

Les processeurs graphiques sont basés sur une architecture parallèle SIMD (*Single Instruction Multiple Data*). Cela signifie qu'une même instruction est exécutée en parallèle sur des données différentes, à la différence du CPU sur lequel, en général, chaque instruction est exécutée sur une seule donnée. Le GPU se montre donc particulièrement efficace pour les tâches lors desquelles on effectue des calculs identiques sur des données indépendantes comme la multiplication de matrices ou le traitement de paquets réseau par exemple.

Le GPU Kepler GK110 a été développé et est fabriqué par la société NVIDIA. Il implémente une architecture CUDA (*compute capability* 3.5) dédiée au GPGPU. Il est constitué de près de 7,1 milliards de transistors pour 15 Multi-Processeurs (MP) dans sa version complète. Ces derniers sont les cœurs opératifs du GPU. Ils sont indépendants les uns des autres et peuvent par conséquent exécuter des tâches différentes. En revanche, au sein de chaque MP, la même instruction est exécutée jusqu'à 32 fois en parallèle. Ces groupes de 32 fils d'exécution, ou *threads*, sont appelés chaînes, ou *warps*. Un MP peut gérer jusqu'à 64 chaînes soit 2048 fils. Au moyen d'un ordonnanceur interne, le MP peut donc mettre en attente des chaînes lorsque celles-ci ne disposent pas encore de toutes les ressources nécessaires pour s'exécuter (après une requête de lecture en mémoire par exemple).

Tableau 4.1 Comparaison des performances (précision simple) entre CPU et GPU

Type	Fabricant	Modèle	Performances
GPU	NVIDIA	K40	4,29 Tflops
		K20	3,52 Tflops
	AMD	Radeon HD 8970	4,30 Tflops
		Radeon HD 7970	4,09 Tflops
CPU	Intel	Xeon E5-2697 v2	518 Gflops
		Xeon E5-2690	371 Gflops
	AMD	Opteron 6284 SE	346 Gflops
		Opteron 6278	333 Gflops

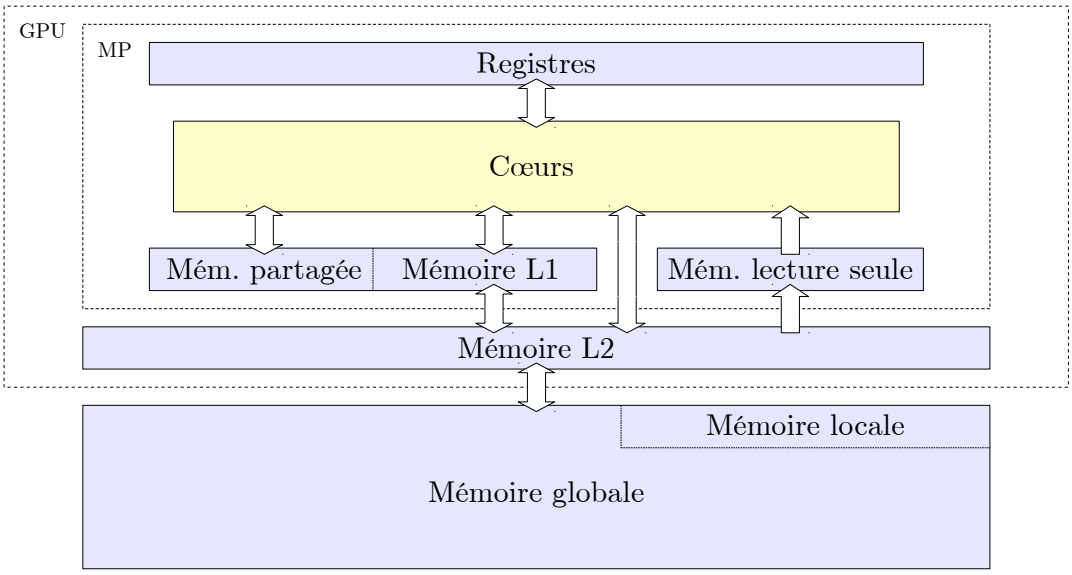

Figure 4.3 Mémoires GPU

Au sein des Multi-Processeurs, du GPU et de la carte dédiée, on trouve différents types de mémoire comme illustré par la figure 4.3. Procédons à une revue de celles-ci :

Registres

Il s'agit de registres de 32 bits alloués aux fils et utilisés pour stocker les données au plus proche des cœurs de calcul. Seulement une dizaine de cycles sont nécessaires pour y accéder. En revanche, les registres sont en nombre limités (65536 pour le GK110) et ils doivent donc être utilisés avec parcimonie.

Mémoire cache L1

Cette mémoire cache de 64 Ko est utilisée pour enregistrer temporairement des données. Il s'agit en général de données accédées récemment ou contigües à celles-ci. Cette mémoire reste très rapide puisque quelques dizaines de cycles seulement sont requis.

Mémoire partagée

La taille de cette mémoire est variable. Elle est en effet installée sur la même mémoire que la cache L1. En fonction de nos besoins, on peut choisir de privilégier soit la L1 soit la mémoire partagée. Les fils d'exécution peuvent utiliser cette espace pour partager des données mais aussi pour s'assurer que celles-ci restent "proches" des cœurs sans pour autant occuper de registres (en nombre limité).

Mémoire cache en lecture seule

Cette mémoire de 48 Ko a une fonction similaire à celle de la cache L1, à ceci près qu'on ne peut qu'y lire des données : pas d'écriture ici.

Mémoire cache L2

Cette mémoire cache, de 768 Ko cette fois-ci, est située en dehors des Multi-Processeurs. Elle est donc partagée entre ceux-ci. Les lignes de cache mesurent ici 128 octets. Cela signifie que lorsqu'une donnée, aussi petite soit-elle, est lue 128 octets seront transférés à la L1 (usage habituel). En revanche, en contournant cette dernière les transferts seront réduis à 32 octets. Lorsque les fils d'une chaîne tentent d'accéder à des données contigües (sur la même ligne de cache), leurs accès seront automatiquement groupés afin de minimiser le nombre de transferts. Cette fonctions est très intéressante puisqu'elle permet de profiter au maximum de chaque lecture dont la latence est plutôt élevée.

Mémoire globale

Il s'agit de la mémoire embarquée sur la carte, aux côtés du GPU. Sa taille varie d'un modèle à l'autre mais compte en général quelques Gigaoctets (Go). Les accès sont relativement lents : 200 à 400 cycles. Son usage est incontournable puisque c'est dans cette mémoire que sont stockées toutes les données transférées depuis le CPU (les paquets réseau dans notre cas).

Mémoire locale

C'est un espace particulier de la mémoire globale dans lequel sont stockées les données qui ne peuvent pas être placées dans les registres ou dans la mémoire partagée. Par le jeu des caches, une donnée sera d'abord placée en L1 mais pourra potentiellement être transférée en L2, voire sur la mémoire globale (mémoire locale), résultant en un accès particulièrement lent.

4.2.2 Programmation logicielle avec CUDA C/C++

Conjointement à son architecture matérielle CUDA, NVIDIA propose une suite logicielle incluant une interface de programmation (API) et des outils pour programmer, compiler, déverminer et profiler des applications pour ses GPU. Les langages de programmation supportés sont le Fortran et le CUDA C/C++ sémantiquement très proche du C/C++.

Le CUDA C/C++ ajoute à la notion de fonction, déjà présente en C/C++, les noyaux qui sont des fonctions exécutées sur GPU. On présente sur la figure 4.4 un exemple de fonction basique et d'un noyau équivalent, ainsi que leurs appels respectifs. La boucle "`for`" pour l'appel de la fonction indique que les N appels se feront séquentiellement. En revanche, concernant le noyau, ses N fils seront traités avec un parallélisme maximal qui dépend de l'architecture et de la charge du GPU. Notons néanmoins que les opérandes devront avoir été copiées vers la mémoire du GPU avant l'appel du noyau. Les préfixes "`h_`" et "`d_`" des pointeurs signifient, par convention, que ceux-ci pointent sur une zone mémoire située respectivement sur l'hôte

(*host*) ou sur la carte GPU (*device*).

Les différents fils d'exécution sont groupés en "blocs" eux-mêmes rassemblés dans une "grille". Une grille est créée à chaque appel d'un noyau et les arguments de cet appel définissent la configuration des blocs et de la grille. Dans l'exemple de la figure 4.4, on a spécifié lors de l'appel "`<<<1, N>>>`" ce qui signifie que l'on aura N fils dans 1 bloc. Les fils s'identifient grâce à leur index "`threadIdx.x`" dans le bloc. Ces configurations de blocs et de fils ne sont pas restreintes à une seule dimension : on peut les déclarer en trois dimensions comme illustré à la Figure 4.5 sur laquelle, par exemple, la grille est composée de $2 \times 2 \times 3 = 12$ blocs chacun composé de $4 \times 4 \times 2 = 32$ fils d'exécution. Un fil s'identifie donc par sa position dans le bloc et la position de celui-ci dans la grille. Lors de l'appel du noyau, le pilote de la carte GPU assignera chaque bloc à un Multi-Processeur qui se chargera alors de son exécution par chaînes (groupes de 32 fils). Il est donc primordial de bien étudier la configuration des grilles et des blocs d'un programme pour occuper au mieux les MP du GPU et d'en maximiser ainsi les performances.

L'appel d'un noyau peut inclure deux autres arguments en plus de ceux configurant la grille et les blocs. Ces nouveaux arguments indiquent le flux auquel appartient l'appel et

```
1  /* Fonction retournant l'addition de deux entiers */
2  void fonction_Addition(int *opA, int *opB, int *resultat, int id){
3    int i = id;
4    resultat[i] = opA[i] + opB[i];
5  }
6
7  /* Noyau equivalent */
8  __global__ void kernel_Addition(int *opA, int *opB, int *resultat){
9    int i = threadIdx.x;
10   resultat[i] = opA[i] + opB[i];
11 }
12
13 int main(){
14 //...
15 //Appel de la fonction pour N additions
16 for(int i=0; i<N; i++)
17   fonction_Addition(h_opA, h_opB, h_resultat, i);
18
19 //Appel du noyau pour N additions
20 kernel_Addition<<<1, N>>>(d_opA, d_opB, d_resultat);
21 //...
22 }
```

Figure 4.4 Fonction en C/C++ et noyau équivalent en CUDA C/C++

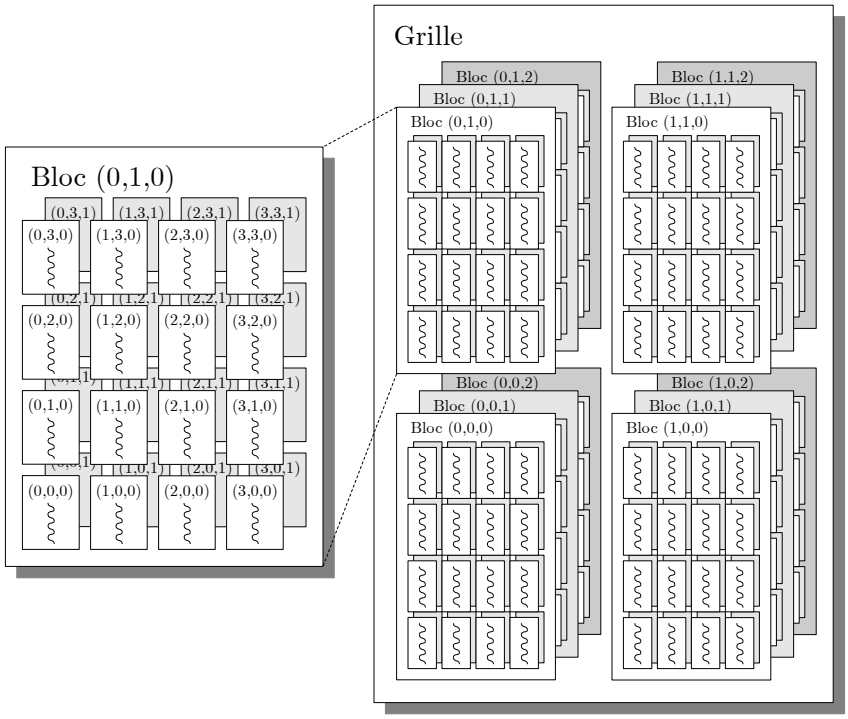

Figure 4.5 Organisation des fils d'exécution, des blocs et de la grille

la taille de mémoire partagée requise pour chaque bloc (en octets). Les flux, ou *streams* en anglais, permettent d'exécuter plusieurs noyaux et de transférer des données depuis et vers le GPU simultanément. On peut donc assigner les tâches (les noyaux et les copies de données) à des flux particuliers afin d'optimiser les traitements et les transferts du GPU. Des tâches appartenant au même flux s'exécuteront séquentiellement tandis que celles appartenant à des flux distincts pourront potentiellement être traitées en parallèle (dépendamment, encore une fois, de l'occupation du GPU). Cet asynchronisme dans le traitement des tâches de différents flux peut être, au besoin, contourné par l'usage des "évènements" (*events*). Un évènement permet de signaler la fin de toutes les tâches dans un flux. Cette information peut alors être utilisée pour lancer des tâches dans d'autres flux ou pour mesurer leurs temps d'exécution.

Les fils d'exécution appartenant à la même chaîne sont traités en parallèle par le MP : ils exécutent la même instruction sur des données potentiellement différentes. Or, dans un programme informatique, il est très courant de recourir aux branchements conditionnels c.à.d quand le chemin d'exécution dépend du résultat d'un test (sur une variable par exemple). Ce chemin peut donc être différent d'un fil à l'autre. Dans un cas pareil, le MP traitera les deux issues possibles du test séquentiellement. Si un grand nombre de divergences sont présentes dans le code alors on peut arriver à une situation dans laquelle les fils sont traités les uns

après les autres sans aucun parallélisme. Dans ce cas extrême, les performances du GPU ne seraient pas meilleures que celles d'un CPU, voire bien moindre. Les divergences sont par conséquent à éviter lors de la rédaction du code d'un noyau.

4.3 Noyaux implémentés

Maintenant que nous avons présenté l'architecture GPU et sa programmation nous allons pouvoir détailler les fonctionnalités des noyaux que nous avons implémentés dans notre système de détection de fichiers et de flux connus. Ces tâches seront appelées après avoir reçu puis copié les paquets Ethernet vers la mémoire globale (embarquée) du GPU.

4.3.1 Analyse des entêtes Ethernet, IP et TCP/UDP

Le rôle de ce noyau est d'analyser les entêtes des paquets stockés dans la mémoire du GPU afin d'en extraire les informations nécessaires lors des tâches suivantes. Il s'agit de déterminer la position et la taille du champ de données (pour le hachage) et d'extraire les adresses IP et les ports TCP/UDP (pour la recherche de flux connu).

Comme nous l'avons indiqué auparavant (section 4.1.2), afin d'optimiser les traitements sur les paquets, ceux-ci sont alignés en mémoire sur des adresses multiples de la taille de "parts" (des octets nuls étant ajoutés comme bourrage). Ces parts seront notamment utiles pour la tâche de hachage. En revanche, pour le noyau actuel cela signifie qu'il faut d'abord identifier sur quelles parts se situent les entêtes des paquets. Lancer un fil d'exécution pour chaque part résulterait, en effet, en une sous-utilisation des Multi-Processeurs puisque seuls quelques fils seraient finalement affectés à un entête. Par conséquent, les positions des entêtes des paquets sont fournies par le CPU qui aura sauvegardé cette information lorsqu'ils stockera les paquets dans sa mémoire tampon.

Lorsque le noyau réalise une lecture des données stockées sur le GPU, des lignes entières de cache sont copiées depuis la mémoire globale vers la L2 puis la L1. Afin de tirer au maximum profit de ces transferts on réalise des lectures aussi grandes que possible (c.à.d de 16 octets). Or, stocker ces 16 octets n'est pas trivial. Les données aussi larges sont, en général, placées en mémoire locale ce qui peut considérablement réduire les performances comme nous l'avons évoqué à la section 4.2.1. On peut en revanche forcer le fait de garder ces données au plus près du processeur en utilisant la mémoire partagée. Chaque fil dispose donc dans cette mémoire d'un espace de 16 octets qui lui est dédié et dans lequel il enregistre ses lectures.

Chaque fil d'exécution assigné à un paquet réalisera les actions suivantes :
- Récupérer la position de l'entête du paquet à traiter.
- Vérifier le champ Ethertype de l'entête Ethernet afin de d'assurer que les données encapsulées utilisent le protocole IPv4.
- Lire le champ "Taille entête" de IPv4 pour déterminer la position de l'entête de la couche transport.
- S'il s'agit bien de TCP ou UDP (champ "Protocole inclus" de IPv4), enregistrer dans la mémoire du GPU la paire d'adresses IPv4 et de ports TCP/UDP pour les noyaux futurs.
- Afin d'éviter que ne soient inutilement hachées une partie de l'entête ou des octets de bourrage, indiquer les zones à hacher pour chaque part qu'occupe le paquet analysé.

Une fois ce noyau terminé, on dispose des informations nécessaires dans la mémoire du GPU pour hacher les données et pour pouvoir identifier si le contexte TCP/UDP-IP du paquet est connu. Ces deux tâches peuvent donc potentiellement être réalisées en parallèle.

4.3.2 Hachage des données des paquets

Ce noyau implémente l'algorithme de *max-Hashing* (voir section 2.3.3). Il consiste en l'extraction d'empreintes à partir des champs de données de chaque paquet. Ces empreintes pourront, dans un autre noyau, être comparées aux empreintes de référence stockées en mémoire. Notre implémentation est semblable à celle réalisée par Lerebours [29] comme nous allons pouvoir le détailler.

Le *max-Hashing* sur GPU

Afin de garantir une originalité suffisante des empreintes, tout en gardant une taille relativement réduite pour celles-ci, la taille de la fenêtre glissant sur les données est fixée à 16 octets (128 bits) et génère des empreintes de 8 octets (64 bits). Le calcul de l'empreinte est donc effectué sur 16 octets et la fenêtre se décalera d'un octet pour l'empreinte suivante : un octet "entre" et un autre "sort" de la fenêtre. Une méthode optimisée permet, pour le calcul d'une nouvelle empreinte, de n'utiliser que les valeurs de l'empreinte précédente et des octets "entrants" et "sortants". Cela permet de ne pas ré-accéder aux 14 autres octets dont les influences sur l'empreinte ont déjà été calculées. Avec h_i l'empreinte calculée à l'étape i, $o_{entrant}$ et $o_{sortant}$ respectivement la valeur des octets entrant et sortant de la fenêtre et la fonction f calculant l'influence de ces octets sur l'empreinte, on peut décrire le calcul optimisé d'une empreinte par l'équation 4.1.

$$h_{i+1} = (h_i \lll 8) \oplus f(o_{entrant}) \oplus f(o_{sortant}) \tag{4.1}$$

Notre fenêtre mesure 16 octets, ce qui signifie que l'influence d'un octet donné doit être conservée dans l'empreinte pendant 16 itérations. Une fois ces 16 itérations effectuées, on aura appliqué $\frac{16 \times 8}{64} = 2$ rotations complètes de l'empreinte (et par conséquent de l'influence de notre octet). Enfin, l'utilisation de l'opérateur `xor` $\oplus$ involutif ($(a \oplus b) \oplus b = a$) nous permet de retirer cette influence de l'empreinte courante.

Les noyaux de calculs d'entiers des GPU actuels étant basés sur une architecture 32 bits, les calculs sont adaptés pour représenter l'empreinte sur deux entiers de 32 bits : h^h et h^b. Ceci revient aux calculs suivants :

$$h^h_{i+1} = [(h^h_i \ll 4) + (h^b_i \gg 28)] \oplus f^h(o_{entrant}) \oplus f^h(o_{sortant}) \tag{4.2}$$

$$h^b_{i+1} = [(h^b_i \ll 4) + (h^h_i \gg 28)] \oplus f^b(o_{entrant}) \oplus f^b(o_{sortant}) \tag{4.3}$$

Enfin, sur un segment donné, l'empreinte qui a la valeur maximale est conservée afin d'être comparée à la base de données. Nous choisissons de garder 4 empreintes : les maxima pour les 4 valeurs possibles des deux bits de poids fort (“00”, “01”, “10” et “11”).

Tâches des fils d'exécution

Chaque paquet occupant un nombre entier de parts et connaissant la taille totale des données transférées au GPU, on est capable de déterminer le nombre total de parts occupées et par conséquent le nombre de fils à lancer. Le noyau précédent ayant déjà analysé l'entête de chaque paquet, il a indiqué, pour chaque fil, la portion à hacher sur la part assignée. Cela exclut donc les entêtes et les zones de bourrage. Les “frontières” des parts doivent aussi faire l'objet d'une attention toute particulière. En effet, puisqu'un même champ de données peut être divisé en plusieurs parts traitées en parallèle, certains fils devront hacher les 15 premiers octets de la part suivante. Toutes les fenêtres possibles auront donc été considérées avec ce recouvrement (figure 4.6).

Les fils sauvegardent en mémoire les 4 empreintes maximales qu'ils auront calculés sur la part qui leur aura été assignée. Ces empreintes pourront alors être utilisées par le noyau suivant c.à.d la comparaison des empreintes calculées sur les données des paquets avec celles préalablement référencées et enregistrées dans la base de données.

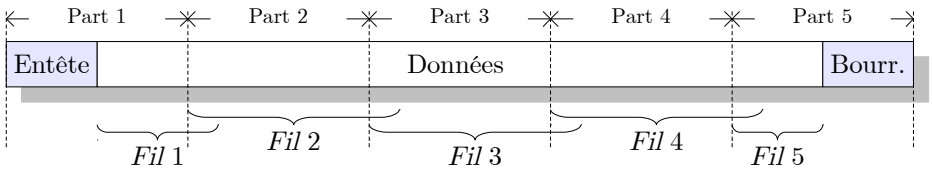

Figure 4.6 Hachage d'un paquet par parts avec recouvrement

4.3.3 Comparaison des empreintes avec la base de données

La base d'empreintes de référence est une table de hachage (voir section 2.3.3). Cette table est constituée de p_1 lignes et c_1 colonnes et chaque cellule peut être occupée par une empreinte (8 octets). Les $log_2(p_1)$ bits de poids faible de l'empreinte constituent l'index de la ligne dans laquelle elle doit être stockée (référencement) ou dans laquelle sont stockées les empreintes de référence auxquelles elle doit être comparée (détection). Les colonnes permettent de stocker dans la même ligne plusieurs empreintes qui ont les mêmes $log_2(p_1)$ bits de poids faible. Nous ne présenterons pas ici le noyau qui initialise la BDD puis ajoute les empreintes dans celle-ci étant donné qu'il ne rentre pas dans le processus de détection à proprement parler. En revanche, il faut mentionner qu'un second tableau est utilisé en parallèle de la table des empreintes. Il est lui aussi constitué de p_1 lignes et c_1 colonnes et permet d'enregistrer un code (sur 32 bits) identifiant le fichier référencé qui a généré chacune des empreintes enregistrées.

Notre noyau devra accéder aux empreintes calculées par le noyau "hacheur" (section 4.3.2) puis les comparer à celles déjà stockées en mémoire. S'il y a égalité entre une empreinte calculée et une empreinte enregistrée dans la table, l'identifiant associé à cette dernière (situé à la même ligne et à la même colonne du second tableau) devra être renvoyé au CPU. On choisit de lancer c_1 fils par empreinte calculée par le hacheur. On peut ainsi profiter des optimisations natives de l'architecture qui groupe les accès à des espaces mémoire adjacents. De plus, on utilise la mémoire partagée pour stocker l'empreinte à comparer, commune à c_1 fils. Cela permet de limiter l'utilisation des registres et de la mémoire locale.

4.3.4 Reconnaissance des flux TCP/UDP-IP

Le but de ce noyau est ici de vérifier si un paquet appartient à un flux à surveiller. On définit un flux à surveiller par une paire d'adresses IP et deux paires de valeurs "limites" pour les ports. Si un paquet possède des adresses IP sauvegardées et que les valeurs de ses ports sont comprises dans les bornes spécifiées alors le paquet doit être signalé au CPU.

L'implémentation de cette fonctionnalité est très proche de celle du noyau détaillé ci-dessus : on dispose d'une table de hachage, de p_2 lignes et c_2 colonnes, et l'index est calculé à partir des deux adresses IP. De la même manière, on lance c_2 fils pour chaque paquet. Par conséquent, pour chaque contexte TCP/UDP-IP extrait il y a un fil assigné à chaque colonne. Ces fils d'exécution se réfèrent donc aux mêmes adresses IP et aux mêmes ports. On notera néanmoins qu'afin de ne pas effectuer c_2 fois le calcul de l'index celui-ci sera effectué par le noyau d'analyse de l'entête (section 4.3.1).

4.3.5 Synthèse

Pour conclure sur les tâches à effectuer pour l'analyse des paquets réseau reçus on note :

- Le transfert des paquets et de leurs positions depuis la mémoire centrale de l'ordinateur vers la mémoire globale du GPU via le bus PCIe.
- L'analyse des entêtes avec l'extraction des informations du flux TCP/UDP-IP (dont calcul de l'index) et le transfert d'indications pour le hachages des parts de chaque paquet.
- Le lancement, en parallèle, des tâches de hachage et de comparaisons avec la base de données de flux TCP/UDP-IP connus.
- La comparaison des empreintes calculées avec celles contenues dans la base de données de référence.
- Le transfert des résultats (concernant les empreintes et les flux) vers la mémoire centrale via le bus PCIe.

Le parallélisme des tâches de hachage et de recherche dans la banque de flux suggère l'utilisation de deux *streams* (voir section 4.2.2). L'un pourra être associé aux transferts, à l'analyse des entêtes, au hachage des paquets et à la recherche dans la base de données d'empreintes, et l'autre sera purement dédié à la recherche des flux TCP/UDP-IP. Un *event* déclenché à la fin de la tâche d'analyse des entêtes permettra de synchroniser le lancement de la tâche du second *stream*.

L'utilisation des bibliothèques de NVIDIA et d'espaces de mémoire non-paginée nous permet simultanément de transférer des données vers et depuis le GPU et d'exécuter des noyaux sur celui-ci. Ainsi, à un temps donné, on pourrait envoyer au GPU les paquets P_{n+1} tout en traitant les paquets P_n et en transférant les résultats pour les paquets P_{n-1}. Le débit total de cette application "*pipelinée*" serait alors fixé par le débit le plus faible : soit celui du PCIe, soit celui du traitement des paquets.

4.4 Résultats pour les transferts sur le PCIe et les noyaux du GPU

Les mesures de temps sont effectuées grâce aux *events* CUDA. Ils sont utilisés pour enregistrer l'occurrence des évènements apparaissant sur le *stream* auquel ils sont rattachés. Pour mesurer le temps nécessaire à l'accomplissement d'une tâche, il suffit simplement de mesurer le temps avant et après celle-ci et de regarder la différence. La figure 4.7 présente un exemple de mesure du temps nécessaire à un transfert et un noyau (rattachés au même *stream*) avec les *events*.

4.4.1 Copie vers et depuis la mémoire embarquée du GPU

Des configurations différentes de transfert ont été testées. Lors du banc de test, on réalise des transferts de taille variable depuis le CPU vers le GPU (H→D). Pour une taille donnée, on effectue plusieurs mesures afin de pouvoir présenter une moyenne représentative. Nos résultats pour le transfert depuis le CPU vers le GPU (H→D) par DMA sont représentés à la figure 4.8 (notez l'échelle logarithmique sur l'axe des abscisses).

Pour tous les transferts inférieurs à 4096 octets (32768 bits), nous avons systématiquement mesuré des temps de l'ordre de 7 μs, et ce, même pour les transferts d'un seul octet. Ce temps représenterait donc la latence des transactions que nous pouvons effectuer sur le bus PCIe-2.0 avec les outils fournis par NVIDIA. La latence ayant peu d'importance dans le cadre de notre système de détection, on peut passer à un point plus critique : la bande passante. On peut voir que l'on dépasse 40 Gb/s à partir de 2×10^6 bits et que l'asymptote, soit 50 Gb/s, est atteinte à partir de 10^7 bits à transférer.

```
1  /* Mesure avant les taches */
2  cudaEventRecord(eventAvant, monStream);
3
4  /* Copie de donnees puis appel d'un noyau */
5  cudaMemcpyAsync(d_memoire, h_memoire, taille, cudaMemcpyHostToDevice,
       monStream);
6  kernel <<< i, j, 0, monStream >>>(d_memoire);
7
8  /* Mesure apres les taches et synchronisation */
9  cudaEventRecord(eventApres, monStream);
10 cudaEventSynchronize(eventFin);
11
12 /* Calcul du temps entre les deux evenements */
13 float temps;
14 cudaEventElapsedTime(&temps, eventAvant, eventApres);
```

Figure 4.7 Mesure du temps avec les *events* CUDA

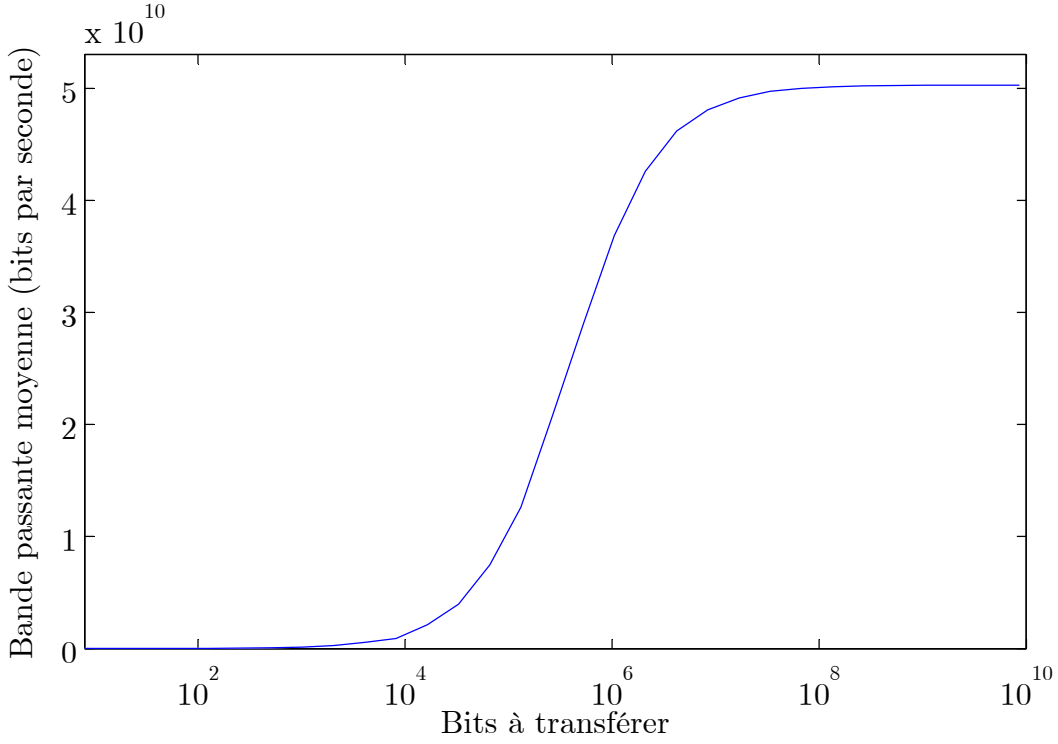

Figure 4.8 Bande passante observée en fonction de la taille des données (H→D)

Le bus PCIe 2.0 peut donc être utilisé dans le cadre d'un système de détection à 40 Gb/s. Nous devrons veiller à ce que la taille de l'espace mémoire tampon soit supérieure à 2×10^6 bits, voire, idéalement, 10^7 bits pour une utilisation optimale du bus. Le système de détection finale utilise un espace mémoire de 64 Mo ($\simeq$ 537 Mbits).

4.4.2 Analyse des entêtes

Nous soumettons un nombre variable de paquets Ethernet au noyau. Il est important de mentionner que les paquets soumis sont complets (entête et champ de données) et qu'ils sont alignés sur des adresses multiples de 256 octets (taille des parts). Par conséquent, les entêtes des paquets ne se trouvent pas sur la même ligne de cache L2 (128 octets), dont on ne peut donc pas tirer parti. Les résultats sont présentés à la figure 4.9.

Nous observons cependant des performances tout à fait satisfaisantes : près de 580 millions d'entêtes peuvent être analysés par le noyau chaque seconde. Dans le pire cas de figure où les paquets Ethernet seraient de taille minimale (64 octets), nous pourrions supporter un flux de $580 \times 10^6 \times 64 \times 8 = 296,96$ Gb/s. Cependant, dans un espace de 64 Mo (comme déterminé précédemment) le nombre maximal de paquets que l'on pourrait soumettre au noyau

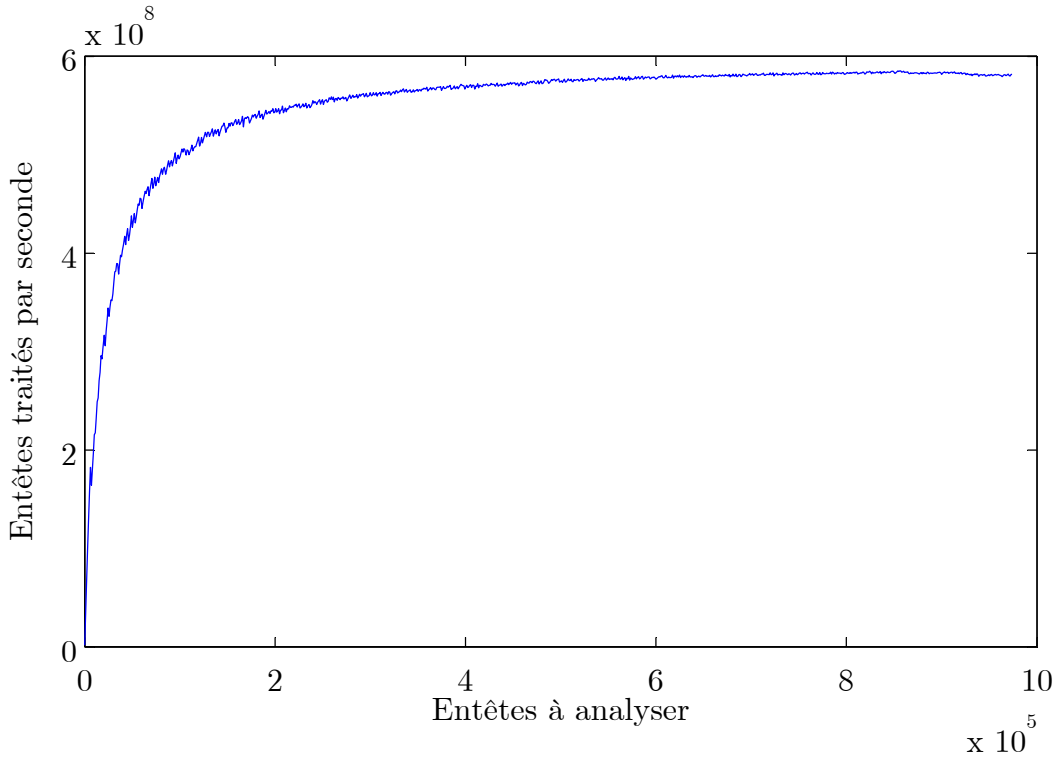

Figure 4.9 Analyse des entêtes

serait alors de l'ordre de 10^5. Pour cette quantité d'entêtes, le noyau est moins performant (seulement 5×10^8 entêtes/seconde) mais ne devrait pas pour autant constituer un goulot d'étranglement. Les performances des autres noyaux et des autres tâches le détermineront.

4.4.3 Recherches dans les bases de données

Les base de données d'empreintes et de flux TCP/UDP-IP connus sont toutes les deux implémentées sous la forme de tables de hachage. Étant donnée cette similarité, nous allons présenter les performances de ces deux tâches de recherche conjointement. Dans le cadre de ces mesures, les données fournies à ces noyaux sont générées de manière aléatoire et ne sont pas triées. Les accès en mémoire seront donc les plus aléatoires possibles. Nous ferons varier le nombre de colonnes dans les bases de données ce qui devrait avoir une influence directe sur les performances. Le nombre de lignes devrait, quant à lui, n'avoir aucune influence (avantage de la table de hachage). D'autre part, deux cas ont été testés : lorsque des correspondances sont trouvées (empreinte ou flux connus) et lorsque les recherches sont infructueuses. Aucune différence notable n'a été observée pour ces différentes issues. Les résultats sont présentés à la figure 4.10.

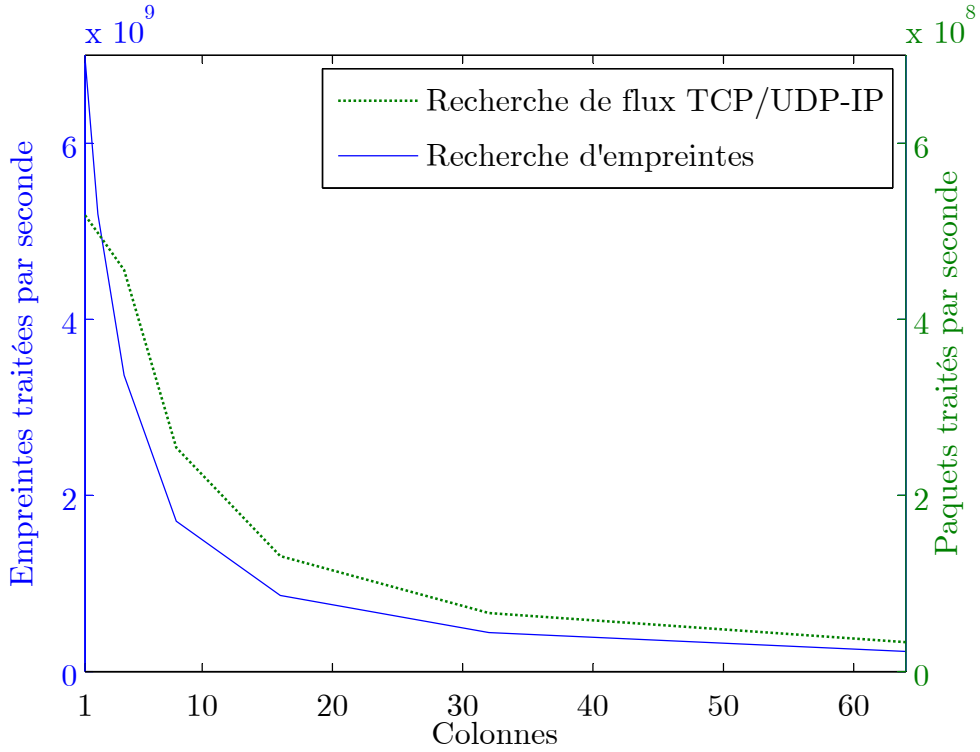

Figure 4.10 Recherche des empreintes et des flux dans les tables

Nous observons la même tendance pour les deux noyaux : le nombre de fils d'exécution lancés étant proportionnel au nombre de colonnes c, les performances sont inversement proportionnelles à c. Toutefois on remarquera que, pour un nombre de colonnes donné, la recherche d'empreintes connues est presque 10 fois plus rapide que celle des flux TCP/UDP-IP. Cela s'explique notamment par la taille des données requises par chacune des tâches et par la complexité de ces dernières. Pour la recherche d'empreintes connues, chaque fil nécessite en effet deux empreintes soit $2 \times 64 = 128$ bits de données et effectue une seule comparaison. Lors de la recherche de flux connus, en revanche, un fil a besoin des informations TCP/UDP-IP d'un paquet (deux adresses IP et deux ports), de l'index correspondant (ici 16 bits), et de la description d'un flux surveillé (deux adresses IP et deux paires de ports). Ceci correspond à un total de $2 \times 32 + 2 \times 16 + 16 + 2 \times 32 + 4 \times 16 = 240$ bits à transférer suivis de 6 comparaisons à effectuer (deux pour les adresses IP et 4 pour les ports).

Sachant qu'à chaque paquet correspond un contexte TCP/UDP-IP à analyser, la tâche de recherche de flux connus présente des performances tout aussi satisfaisantes que celles de l'analyse des entêtes. Par conséquent, celle-ci ne devrait pas non plus constituer un goulot d'étranglement pour notre application (avec $c < 64$ en tout cas).

En ce qui concerne la recherche d'empreintes connues, on génère 4 empreintes par "part" de paquet. La taille de ces parts est donc importante puisqu'elle influe directement sur le nombre total d'empreintes qui seront générées. Une grande taille réduit le nombre d'empreintes générées et, par conséquent, limite le temps requis par la tâche de recherche dans la base de données. En revanche, puisqu'une fenêtre générant l'empreinte maximale localement garde cette propriété même pour un voisinage plus réduit mais peut perdre cette propriété pour un voisinage plus grand, alors plus le nombre d'empreintes à générer est grand (plus la taille des voisinages considérés est petite), plus le taux de faux-négatif est réduit.

4.4.4 Hachage des champs de données

Notre banc de test consiste à calculer des empreintes sur des données générées aléatoirement. On fait varier le nombre de fils d'exécution à lancer en soumettant un nombre croissant de données. La taille des parts est ici fixée à 256 octets ce qui signifie, en considérant le recouvrement, que chaque fil hache $256 + 15 = 271$ octets et en extrait 4 empreintes. Il s'agit donc de 15 octets qui seront lus deux fois (réutilisés). La taille des parts a là encore une influence puisque le taux de réutilisation des données diminue lorsque la taille des parts augmente.

On considère dans un premier temps l'implémentation de l'algorithme tel que spécifié à la section 4.3.2, c.à.d avec une fonction de hachage opérant sur des données représentées sur 32 bits. Dans ce cas-ci, les octets sont lus et hachés les uns après les autres tout en vérifiant, à chaque itération, si l'empreinte générée est un maximum. L'issue de cette vérification n'étant probablement pas la même pour tous les fils cela introduira de la divergence dans le chemin d'exécution de nos fils. Il s'agit là du principal défaut de l'implémentation de l'algorithme de max-Hashing sur GPU. On parvient néanmoins à atteindre 94 Gb/s (figure 4.13, tracé bleu continu intitulé "Sans filtrage").

```
1 if( a == CARAC_0 || a == CARAC_1 || a == CARAC_2 ||
2   a == CARAC_3 || a == CARAC_4 || a == b){
3   ... // Octet a filtrer, passons a l'octet suivant
4 }
5 else{
6   ... // Octet correct, calculons l'empreinte avec celui-ci
7 }
```

Figure 4.11 Filtre sommaire de caractères (octets)

Nous introduisons à présent un filtrage des octets lus. Celui-ci a pour objectif de ne pas considérer, lors de la génération des empreintes, les octets ayant une valeur égale à certains caractères de contrôle ASCII et les octets consécutifs de même valeur. Notons “a” la valeur de l’octet entrant et “b” la valeur de l’octet lu précédemment. Sachant que l’on souhaite filtrer certains caractères et les répétitions de caractère identiques, la première mise en place de notre filtre est semblable au code représenté à la figure 4.11 (les valeurs des 5 caractères sont notées “`CARAC_0`” à “`CARAC_4`”). Ce genre de filtre ajoute inévitablement de la divergence dans l’exécution. Certains fils liront un octet qui devra être filtré et d’autres liront un octet qui devra, celui-ci, être haché. L’inconvénient de cette première implémentation du filtre c’est la manière dont il sera compilé. Une divergence sera introduite pour chaque contrôle soit donc 7 chemins d’exécution en tout : une pour chaque caractère à filtrer et une pour les caractères non filtrés. Ces multiples divergences réduisent les performances du hacheur de près de 50%, comme on peut le voir sur la figure 4.13 (courbe rouge à tirets intitulée “Avec filtrage basique”) soit 48 Gb/s.

Une nouvelle implémentation de ce filtre est envisagée afin de retrouver de meilleures performances. Nous choisissons d’ajouter de nouvelles instructions afin de réduire les divergences induites par le filtre. Le code de la nouvelle implémentation est visible à la figure 4.12. On se sert simplement de la propriété de la multiplication qui veut que si au moins un des facteurs est nul alors le produit de tous les facteurs est nul. Les performances atteintes avec cette implémentation sont observables sur la figure 4.13 (courbe verte en pointillés intitulée “Avec filtrage amélioré”). On obtient une vitesse de traitement capable de supporter 71 Gb/s soit 24% de moins que le hacheur sans filtrage.

```
1  float f_a = (float)a;
2
3  f_a = (f_a - CARAC_0) * (f_a - CARAC_1) * (f_a - CARAC_2) *
4      (f_a - CARAC_3) * (f_a - CARAC_4) * (f_a - ((float)b));
5
6  if(f_a == 0){
7    ... // Passons a l'octet suivant
8  }
9  else{
10   ... // Calculons l'empreinte avec cet octet
11 }
```

Figure 4.12 Filtre amélioré de caractères (octets)

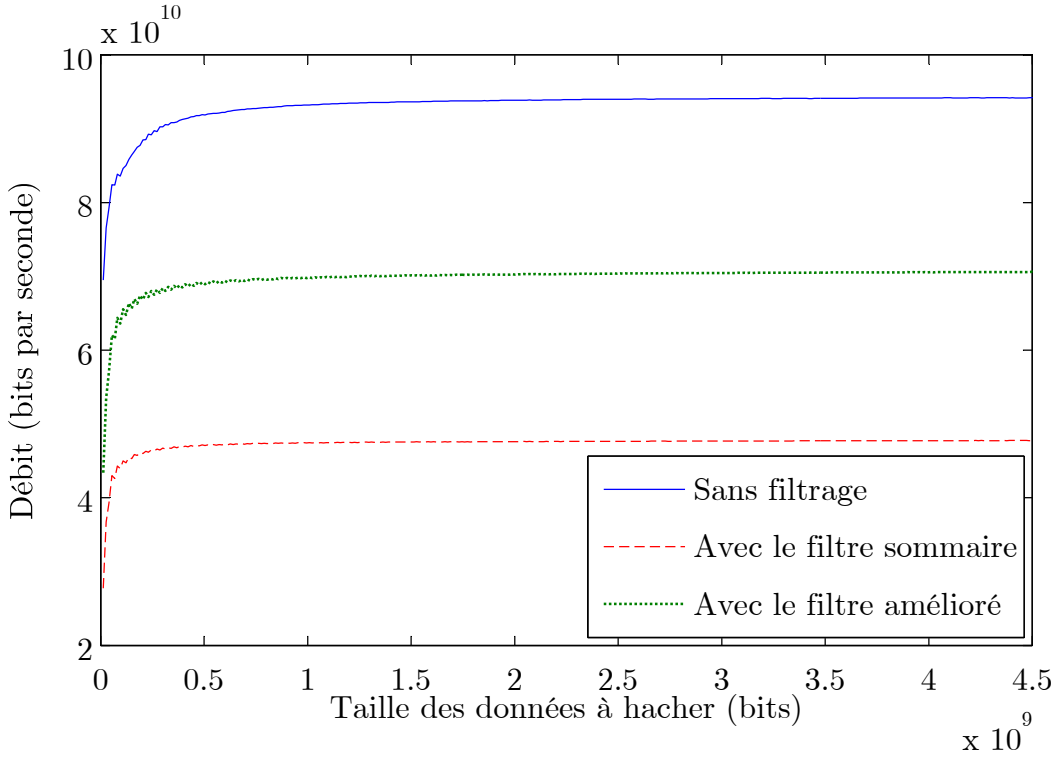

Figure 4.13 Performances du hacheur dans 3 configurations différentes

4.5 Système complet et *pipeline*

Maintenant que les performances de chaque noyau ont été mesurées indépendamment, on souhaite observer le comportement du système complet. On mesure donc le temps nécessaire pour le traitement complet de paquets Ethernet réels. Ce traitement consiste tout d'abord en la réinitialisation des zones de stockage des résultats dans la mémoire du GPU (tâche non discutée dans ce document), en l'analyse des entêtes des paquets, puis en parallèle le hachage des champs de données et la recherche des flux TCP/UDP-IP connus et enfin, en la recherche des empreintes connues dans la base de données. L'enchainement de toutes les tâches effectuées sur le GPU est schématisé sur la figure 4.14.

Nous conservons le hacheur doté du filtre amélioré et fixons la largeur des bases de données à 8 colonnes pour celle concernant les flux et 16 colonnes pour celle des empreintes. Les performances mesurées en fonction de la taille des données soumises sont présentées à la figure 4.15. On observe que l'on parvient à atteindre une vitesse de traitement de 52 Gb/s (courbe continue bleue), que l'on peut comparer aux 50 Gb/s atteints par le transfert des paquets sur le bus PCIe 2.0 (courbe rouge en pointillés). Avec au moins 64 Mo de données soumis au système ($\simeq 5 \times 10^8$ bits), on s'assure que le transfert de celles-ci sera plus lent que leur traitement.

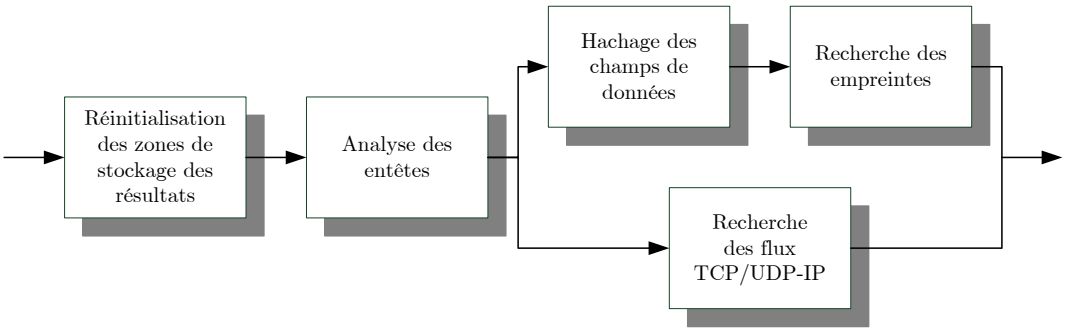

Figure 4.14 Enchainement des noyaux

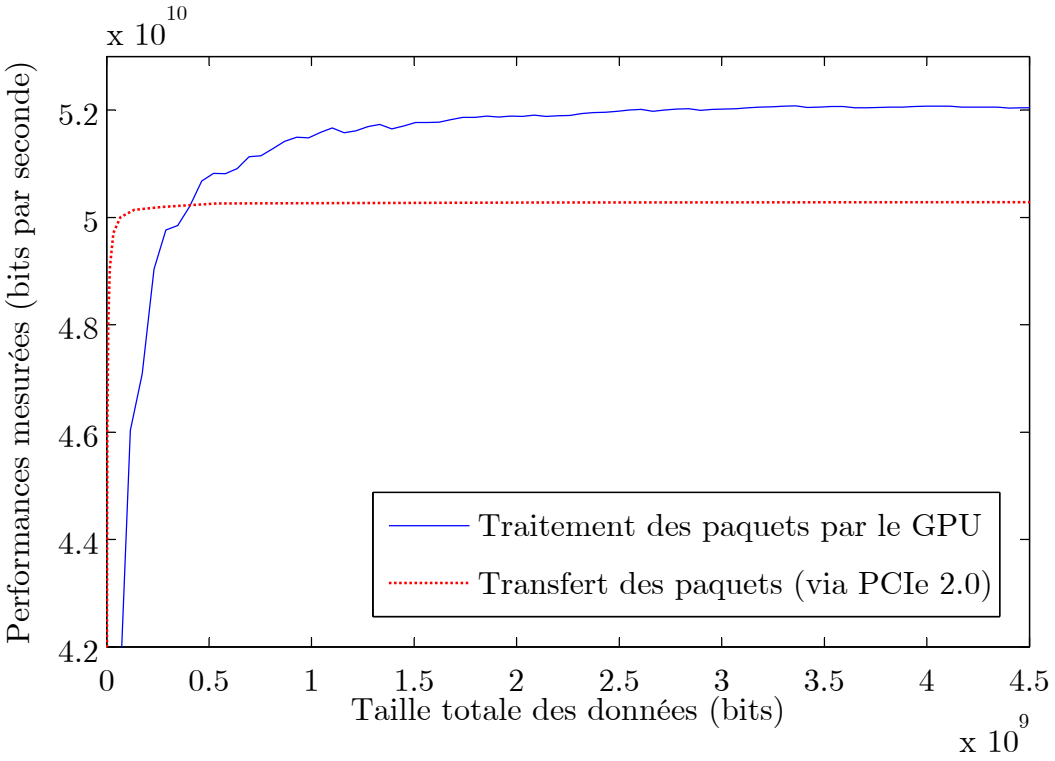

Figure 4.15 Performances de toute la partie de traitement (noyaux)

Néanmoins, sans parallélisme des transferts et des traitement on ne peut supporter qu'un flux de données d'environ 25 Gb/s. Or, comme nous l'avons vu auparavant, en utilisant un espace de mémoire non-paginée on peut simultanément utiliser le bus PCIe 2.0 dans les deux sens (CPU vers GPU et GPU vers CPU) et réaliser des traitements sur le GPU et sur le CPU. En considérant le bus PCIe comme le goulot d'étranglement de notre système, ce dernier ne pourra, par conséquent, jamais supporter un débit supérieur à 50 Gb/s. Nous souhaiterions

donc qu'à chaque instant les paquets P_{n+1} soient envoyés à la mémoire du GPU, que les paquets P_n soient traités sur le GPU, et que les résultats des paquets P_{n-1} soient envoyés au CPU puis traités par celui-ci. Les paquets passent donc par trois grandes étapes dans le système.

Sur la figure 4.16, nous présentons le principe du "pipelinage" de notre système. Trois contextes différents sont nécessaires (*streams* 13-14, 15-16 et 17-18) pour, d'une part, utiliser le bus PCIe au maximum et, d'autre part, pour laisser le temps au CPU de traiter les résultats renvoyés par le GPU. Chaque contexte passe bien par trois grandes étapes : le transfert des paquets ("*Memcpy*") vers la mémoire du embarquée sur GPU, le traitement des paquets par les différents noyaux puis enfin le transfert vers le CPU et le temps "libre" pour le traitement des résultats sur celui-ci. L'outil de profilage nous permet de grouper toutes ces actions en fonction des ressources qu'elles utilisent (PCIe ou GPU), on présente une telle vue sur la Figure 4.17. Nous observons que le bus PCIe, dans le sens du CPU vers le GPU ("*Memcpy (HtoD)*"), est utilisé près de 100% du temps. Quant au GPU, il dispose de moments de répit mettant en avant la marge entre les performances du transferts et celles des traitements.

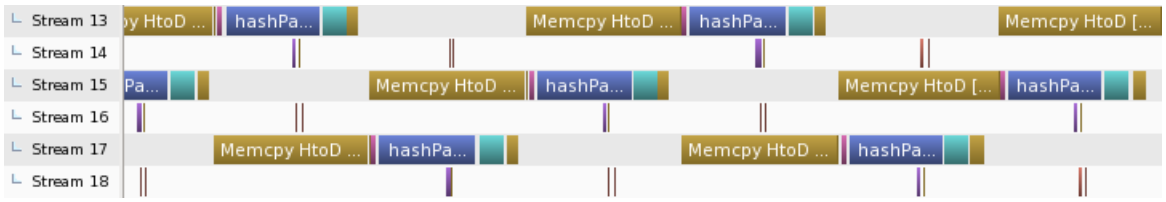

Figure 4.16 Copie d'écran du profileur NVIDIA (1) : étages du *pipeline*

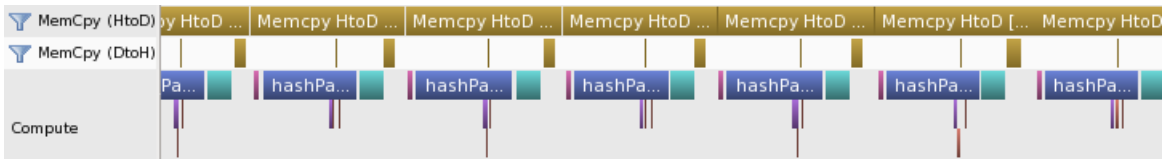

Figure 4.17 Copie d'écran du profileur NVIDIA (2) : occupations du bus et du GPU

4.5.1 Conclusion du chapitre

Au cours de ce chapitre, nous avons présenté notre système de détection de fragments de fichiers connus et de flux TCP/UDP-IP connus. Nous avons d'abord effectué une brève présentation des subtilités de l'architecture et de la programmation des processeurs graphique (GPU), notamment les différents niveaux de mémoire et le parallélisme des fils d'exécution (non-divergence) au sein d'une même chaîne. Partant de ces remarques, les tâches de chaque noyau exécuté sur GPU ont été présentées ainsi que les performances de nos implémentations. Ces noyaux analysent le contenu de l'entête des paquets, hachent les champs de données et recherchent dans une base de données embarquée dans la mémoire du GPU une correspondance avec des empreintes connues et des flux TCP/UDP-IP connus. L'exécution de nos noyaux sur GPU nous permettent de traiter jusqu'à 52 Gb/s de paquets réseau tandis que le bus PCIe 2.0, lui, limite notre système à 50 Gb/s. Le système complet *pipeliné* permet donc de traiter jusqu'à 50 Gb/s pour la recherche de fragments de fichiers connus et de paquets appartenant à un flux TCP/UDP-IP connu.

CHAPITRE 5

CONCLUSION

5.1 Synthèse des travaux

Au cours de ce mémoire, nous avons présenté un système complet de référencement de fichiers vidéo et de détection de fichiers connus et de flux connus sur un lien réseau. Pour le référencement, une étude approfondie des spécifications des formats MP4 et H.264 nous a permis de localiser les zones à "haute-entropie" dans les fichiers vidéo utilisant ces standards. La tâche de détection de fichiers et de flux connus est principalement déportée sur GPU afin de profiter de la puissance de calcul de ces plateformes et dans la continuation des travaux de Lerebours [29].

Nous avons pu observer que le référencement brut de fichiers vidéo avec l'algorithme de *max-Hashing* génère de nombreuses redondances dans les empreintes calculées. L'impact de ces redondances se retrouve notamment dans l'apparition de faux-positifs dans l'étape de détection. En nous concentrant sur les zones dites à "haute-entropie" des fichiers vidéo, nous sommes parvenus à prévenir très efficacement l'apparition de telles redondances ne générant donc plus aucun faux-positif dans le cadre de nos tests et remplissant ainsi notre premier objectif de recherche.

Concernant la partie détection, et notre second objectif de recherche, les traitements des paquets consistent en l'analyse de leurs entêtes, la recherche des flux TCP/UDP-IP dans une base de données (BDD), le calcul d'empreintes sur leurs champs de données (avec l'algorithme de *max-hashing*) et la recherche de ces empreintes dans une BDD. Les performances de ces tâches implémentées sur le GPU permettraient de supporter un débit total de 52 Gb/s. Mais, au regard des 50 Gb/s qu'atteignent les transferts via le bus PCIe-2.0, nous avons mis en place un système *pipeliné* limité uniquement par les transactions sur le bus et permettant de traiter jusqu'à 50 Gb/s de paquets Ethernet.

Outre la puissance de traitement obtenue grâce au GPU, nous profitons aussi des avantages inhérents à l'algorithme de *max-Hashing*. Citons tout d'abord que, avec l'utilisation d'une table de hachage et une fois la taille de celle-ci fixée, les performances du système ne seront pas amoindries lors de l'ajout de nouvelles "règles" (l'ajout de nouvelles empreintes

ou de nouveaux flux à détecter). Il s'agit d'un avantage sur les algorithmes d'expressions régulières qui voient leurs performances diminuer avec l'accroissement du nombre de règles. D'autre part, les empreintes de 8 octets générées à partir de fenêtres de 16 octets représentent une section de fichier sans pour autant divulguer celle-ci. On peut ainsi distribuer les empreintes à détecter (représentant par exemple des fichiers illégaux) sans distribuer ces fichiers ou une partie de ceux-ci.

Les outils de référencement dédiés aux fichiers vidéo présentés dans ce mémoire peuvent être utilisés par toute personne ou organisme ayant accès à des données sensibles dont la propagation n'est pas souhaitée ou constitue un délit. Citons par exemple les forces de police, les ayant-droits d'œuvres protégées par le droit d'auteur ou encore les entreprises soucieuses de ne pas voir certains fichiers être diffusés en dehors de leurs réseaux. Les performances atteintes par le système de détection sur GPU, soit 50 Gb/s, nous permettent de cibler les réseaux de grandes entreprises, ceux de fournisseurs d'accès à Internet voir même les dorsales Internet.

5.2 Limitation de la solution proposée

La limitation principale du système complet décrit dans ce mémoire se situe à l'étape de transfert des paquets entre l'interface réseau et la mémoire de notre application CPU (voir 4.1.1). Lerebours [29] avait déjà déterminé que les performances de la bibliothèque `Pcap` [57] avec un système d'exploitation GNU/Linux natif ne permettait pas de capturer plus de 7,3 Gb/s. Le recours à la bibliothèque `PF_RING` [39], tel que suggéré par Lerebours, permit de capturer près de 15 Gb/s de données sur 35 Gb/s qui ont pu être envoyés. Même si ces résultats constituent une progression, notre implémentation ne permet pas de fournir au reste du système ni les 50 Gb/s de données qu'il est capable de traiter, ni même les 40 Gb/s que notre interface réseau peut recevoir.

5.3 Améliorations futures

5.3.1 Capture des paquets

Notre implémentation de `PF_RING` n'ayant pas eu des performances suffisantes, nous pourrions nous ré-orienter vers d'autres bibliothèques de traitement rapide des paquets réseau. Dans cette catégorie, la bibliothèque `DPDK` [12] pourrait s'avérer prometteuse pour notre système compte-tenu du support des processeurs Intel Xeon et des interfaces réseaux Mellanox que nous possédons déjà. D'autre part, nous pourrions imaginer remplacer la carte réseau

que nous utilisons actuellement par une carte FPGA dotée d'une interface réseau. Certains constructeurs annoncent en effet que de telles configurations permettent de capturer et de transmettre des paquets aux applications à un débit de 10, 40 et 100 Gbps [6].

5.3.2 Extension du référencement dédié

La méthode présentée pour déterminer et extraire les zones à "haute-entropie" des fichiers MP4 avec H.264 peut être étendue à d'autres formats. Il serait notamment intéressant de considérer les flux audios utilisant le codec populaire AAC (*Advanced Audio Coding*, souvent utilisé conjointement à H.264). La prise en charge d'autres formats vidéos, audios et d'autres conteneurs pourraient aussi être mise en place. Par exemple, le conteneur WEBM utilisant le codeur vidéo VP8 est quelques fois présenté comme le futur format "standard" pour HTML5. Enfin, notre méthode pourrait être étendue aux fichiers non-multimédia qui comportent, eux aussi, des zones à "entropie" variable.

5.3.3 Bus PCIe 3.0

Les performances du système de détection présenté dans ce mémoire sont d'abord limitées par la bande passante du bus PCIe 2.0. Celle-ci est théoriquement de 64 Gb/s, mais nous n'avons pu observer de débits supérieurs à 50 Gb/s. En nous réorientant vers une carte mère comportant un bus PCIe 3.0 et une carte GPU compatible avec celui-ci, nous pourrions atteindre une bande passante supérieure à 100 Gb/s pour les transferts entre le CPU et le GPU (128 Gb/s théoriques). Toutes choses étant par ailleurs égales, les traitements constitueraient alors le goulot d'étranglement avec une capacité de 52 Gb/s seulement.

5.3.4 Autres GPU

Plusieurs solutions s'offrent à nous pour l'amélioration des performances du traitement des paquets. La première, et la plus simple, consisterait à tester d'autres GPU. Idéalement, nous devrions rester avec le même fabricant, NVIDIA, afin de conserver les codes en CUDA C/C++ déjà rédigés (seule l'étape de compilation devra être adaptée). Nous pourrions alors nous diriger vers une version plus récente et plus performante de notre GPU dédié aux calculs généralistes (ex : K40, voir tableau 4.1). Nous avons aussi la possibilité de changer de gamme de GPU et de nous orienter vers les produits dédiés à la 3D et aux jeux-vidéo. Outre leurs prix plus abordables, ces GPU disposent en général d'accès en mémoire plus rapides et de performances de calcul, en simple précision, plus importantes : 4,29 Tflops pour le Tesla K40 (GPGPU) contre 5 Tflops pour le Geforce GTX980 (GPU dédié à la 3D et aux jeux-vidéo). Ces avantages sont à considérer au regard d'une durabilité généralement moindre.

5.3.5 Solution à multiple GPU

En dotant notre système de plusieurs GPU, nous pourrions répartir les tâches de traitement entre ceux-ci. Le hachage, notamment, pourrait bénéficier de ces multiples périphériques étant la tâche la plus longue de notre système pour une quantité de paquets donnée. Sa lenteur est principalement causée par la quantité de données à transférer et à traiter. La capacité totale de stockage serait, de surcroit, décuplée et permettrait de conserver plus d'empreintes de référence et plus d'informations de flux TCP/UDP-IP à surveiller. En revanche, la latence totale du système augmentera notablement puisqu'il faudra alors considérer les transferts de GPU à GPU.

Une implémentation plus simple et plus directe consisterait à diviser les paquets Ethernet reçus en autant de groupes qu'il y a de GPU à disposition. Tous les GPU lanceraient alors les mêmes noyaux pour traiter leurs paquets. Un désavantage résidera dans le stockage des bases de données d'empreintes et de flux : elles devront être dupliquées sur les mémoires de tous les GPU.

RÉFÉRENCES

[1] A. Adesina, H. Nyongesa, and K. Agbele. Digital watermarking : A state-of-the-art review. In *IST-Africa, 2010*, pages 1–8, May 2010.

[2] A. V. Aho and M. J. Corasick. Efficient string matching : an aid to bibliographic search. *Commun. ACM*, 18(6) :333–340, 1975.

[3] F. Ahrens. Hollywood says piracy has ripple effect. *The Washington Post*, 2006. http ://www.washingtonpost.com/wp-dyn/content/article/2006/09/28/AR2006092801640.html [Accédé en novembre 2014].

[4] S. N. Biswas, T. Hasan, S. Dasgupta, S. R. Das, V. Groza, E. M. Petriu, and M. H. Assaf. Compressed video watermarking technique. In *Instrumentation and Measurement Technology Conference (I2MTC), 2013 IEEE International*, pages 1790–1794, 2013.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10) :762–772, 1977.

[6] CESNET. Cesnet and invea-tech demonstrate 100 gbps transfers over pcie with a single fpga. 2014. http ://www.cesnet.cz/cesnet/reports/press-releases/100-gbps-over-pcie-with-single-fpga/ ?lang=en [Accédé en novembre 2014].

[7] S. Chase and C. Freeze. Hollywood says piracy has ripple effect. *The Globe And Mail*, 2014. http ://www.theglobeandmail.com/news/national/chinese-hacked-government-computers-ottawa-says/article19818728/ [Accédé en novembre 2014].

[8] L. Cheng-Hung, L. Chen-Hsiung, C. Lung-Sheng, and C. Shih-Chieh. Accelerating pattern matching using a novel parallel algorithm on gpus. *Computers, IEEE Transactions on*, 62(10) :1906–1916, 2013.

[9] Cisco. Vni forecast highlights, 2012. http ://www.cisco.com/assets/sol/sp/vni/forecast_ highlights_mobile/index.html [Accédé en septembre 2014].

[10] ClamAV. Clam antivirus, 2002. http ://www.clamav.net/index.html [Accédé en septembre 2014].

[11] J. P. David. Max-hashing fragments for large data sets detection. In *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, pages 1–6, 2013.

[12] DPDK. Data plane development kit. http ://dpdk.org/.

[13] D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

[14] Envisional. An estimate of infringing use of the internet. Technical report, 2011. http ://documents.envisional.com/docs/Envisional-Internet_Usage-Jan2011.pdf [Accédé en novembre 2014].

[15] FFmpeg, 2014. https ://www.ffmpeg.org/.

[16] A. Frei, N. Erenay, V. Dittmann, and M. Graf. Paedophilia on the internet–a study of 33 convicted offenders in the canton of lucerne. *Swiss Med Wkly*, 135(33-34) :488–94, 2005.

[17] A. A. Ghorbani, W. Lu, M. Tavallaee, and SpringerLink. *Network Intrusion Detection and Prevention : Concepts and Techniques*, volume 47. Springer US, Boston, MA, 2010.

[18] N. Harbour. dcfldd, 2002. http ://dcfldd.sourceforge.net/ [Accédé en septembre 2014].

[19] R. N. Horspool. Practical fast searching in strings. *Software : Practice and Experience*, 10(6) :501–506, 1980.

[20] A. Husagic-Selman, R. Koker, and S. Selman. Intrusion detection using neural network committee machine. In *Information, Communication and Automation Technologies (ICAT), 2013 XXIV International Symposium on*, pages 1–6, 2013.

[21] G. Inc. Fonctionnement de content id. https ://support.google.com/youtube/answer/2797370 ?hl=fr [Accédé en novembre 2014].

[22] ITU-T. *ITU-T Recommendations for H.264*. 2014.

[23] Y.-G. Jiang, G. Ye, S.-F. Chang, D. Ellis, and A. C. Loui. Consumer video understanding : A benchmark database and an evaluation of human and machine performance. In *Proceedings of ACM International Conference on Multimedia Retrieval (ICMR), oral session*, 2011.

[24] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2) :249–260, 1987.

[25] R. A. Kemmerer and G. Vigna. Intrusion detection : a brief history and overview. *Computer*, 35(4) :27–30, 2002.

[26] D. E. Knuth, J. J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2) :323, 1977.

[27] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplement :91–97, 2006.

[28] J. Kornblum. Ssdeep, 2006. http ://ssdeep.sourceforge.net/ [Accédé en septembre 2014].

[29] J. Lerebours. *Filtrage de contenus numeriques connus a haute vitesse optimise sur plateforme GPU*. PhD thesis, 2012.

[30] M. S. Lew, N. Sebe, C. Djeraba, and R. Jain. Content-based multimedia information retrieval : State of the art and challenges. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2(1) :1–19, 2006.

[31] Libav, 2014. https ://libav.org/.

[32] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7) :620–636, 2003.

[33] L. Marziale, G. G. Richard Iii, and V. Roussev. Massive threading : Using gpus to increase the performance of digital forensics tools. *digital investigation*, 4, Supplement(0) :73–81, 2007.

[34] R. C. Merkle. *Secrecy, authentication, and public key systems.* PhD thesis, 1979.

[35] L. Monde. Loppsi 2 : les "sages" valident le blocage des sites pédo-pornographiques. *Le Monde*, 2011. http ://www.lemonde.fr/technologies/article/2011/03/11/loppsi-2-les-sages-valident-le-blocage-des-sites-pedo-pornographiques_1491526_651865.html [Accédé en novembre 2014].

[36] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1) :31–88, 2001.

[37] T. Nhat-Phuong, L. Myungho, H. Sugwon, and S. Minho. Memory efficient parallelization for aho-corasick algorithm on a gpu. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 432–438.

[38] G. Niveau. Cyber-pedocriminality : Characteristics of a sample of internet child pornography offenders. *Child Abuse & Neglect*, 34(8) :570–575, 2010.

[39] NTOP. Pf_ring, 2014. http ://www.ntop.org/products/pf_ring/.

[40] NVIDIA. Cuda c programming guide version 6.5, 2014. http ://docs.nvidia.com/cuda/cuda-c-programming-guide/ [Accédé en novembre 2014].

[41] N. I. of Standards and Technology. National software reference library, Aug. 2003. http ://www.nsrl.nist.gov/ [Accédé en septembre 2014].

[42] S. Paschalakis, K. Iwamoto, P. Brasnett, N. Sprljan, R. Oami, T. Nomura, A. Yamada, and M. Bober. The mpeg-7 video signature tools for content identification. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(7) :1050–1063, 2012.

[43] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee. Mcpad : A multiple classifier system for accurate payload-based anomaly detection. *Comput. Netw.*, 53(6) :864–881, 2009.

[44] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2) :114–125, 1959.

[45] RIAA. Frequently asked questions, 2014. http ://www.riaa.com/faq.php [Accédé en novembre 2014].

[46] I. E. Richardson. *H. 264 and MPEG-4 video compression : video coding for next-generation multimedia.* John Wiley and Sons, 2004.

[47] R. Rivest. Md5, 1992. https ://www.ietf.org/rfc/rfc1321.txt [Accédé en septembre 2014].

[48] M. Roesch. Snort - lightweight intrusion detection for networks, 1999.

[49] V. Roussev. Hashing and data fingerprinting in digital forensics. *Security and Privacy, IEEE*, 7(2) :49–55, 2009.

[50] V. Roussev. *Data Fingerprinting with Similarity Digests*, volume 337/2010, pages 207–226. Springer, 2010.

[51] V. Roussev. An evaluation of forensic similarity hashes. *Digital Investigation*, 8(SUPPL.) :S34–S41, 2011.

[52] V. Roussev and S. L. Garfinkel. File fragment classification-the case for specialized approaches. In *Systematic Approaches to Digital Forensic Engineering, 2009. SADFE '09. Fourth International IEEE Workshop on*, pages 3–14, 2009.

[53] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing : local algorithms for document fingerprinting, 2003.

[54] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating gpus for network packet signature matching. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 175–184. IEEE, 2009.

[55] I. Standard. Iso/iec 14496-12 : Iso base media file format, 2012.

[56] I. Standard. Iso/iec 14496-10 : Advanced video coding, 2013-04-13 2013.

[57] TCPDump. libpcap, 2014. http ://www.tcpdump.org/.

[58] K. Thompson. grep, 1973. http ://www.gnu.org/software/grep/.

[59] A. Tridgell. spamsum, 2002. http ://www.samba.org/ftp/unpacked/junkcode/spamsum/ [Accédé en septembre 2014].

[60] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort : High performance network intrusion detection using graphics processors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 116–134, 1433016, 2008. Springer-Verlag.

[61] G. Vasiliadis and S. Ioannidis. *GrAVity : A Massively Parallel Antivirus Engine - Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 79–96. Springer Berlin / Heidelberg, 2010.

[62] Verizon-Business. Network facts. http ://www.verizon.com/about/sites/default/files/ Verizon_Fact_Sheet.pdf [Accédé en novembre 2014].

[63] W3C, 2014. http ://www.w3.org/.

[64] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full sha-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer, 2005.

[65] R. Wortley and S. Smallbone. Child pornography on the internet, 2006.

[66] T. Xie, F. Liu, and D. Feng. Fast collision attack on md5. *IACR Cryptology ePrint Archive*, 2013 :170, 2013.

[67] Youtube. Statistics, 2014. https ://www.youtube.com/yt/press/fr-CA/statistics.html [Accédé en septembre 2014].