View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by PolyPublie

UNIVERSITÉ DE MONTRÉAL

# IMPLÉMENTATIONS LOGICIELLE ET MATÉRIELLE DE L'ALGORITHME AHO-CORASICK POUR LA DÉTECTION D'INTRUSIONS

ALEXSANDRE BONNEAU LACROIX

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

DÉCEMBRE 2016

© Alexsandre Bonneau Lacroix, 2016.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

IMPLÉMENTATIONS LOGICIELLE ET MATÉRIELLE DE
L'ALGORITHME AHO-CORASICK POUR LA DÉTECTION D'INTRUSIONS

présenté par : LACROIX Alexsandre Bonneau

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. DAGENAIS Michel, Ph. D., président

M. LANGLOIS J.M. Pierre, Ph. D., membre et directeur de recherche

M. BOIS Guy, Ph. D., membre et codirecteur de recherche

M. PIERRE Samuel, Ph. D., membre

# DÉDICACE

*Tout ce qui mérite d'être fait mérite d'être bien fait.*

# REMERCIEMENTS

J’aimerais remercier mon directeur de recherche, Pierre Langlois, ainsi que mon codirecteur, Guy Bois, pour toutes les discussions, les conseils, les échanges de connaissances et les échanges d’idées. Tout au long de ma maîtrise, mon directeur a su me faire progresser afin d’atteindre des objectifs que je ne pensais pas initialement possibles. De plus, j’aimerais remercier tous les membres de mon laboratoire pour les discussions et les échanges d’idées. Finalement, j’aimerais aussi remercier mes proches qui m’ont soutenu et motivé pendant ces deux dernières années.

# RÉSUMÉ

Ce travail propose des méthodes et architectures efficaces pour l'implémentation de l'algorithme Aho-Corasick. Cet algorithme peut être utilisé pour la recherche de chaînes de caractères dans un système de détection d'intrusion, tels que Snort, pour les réseaux informatiques.

Deux versions sont proposées, une version logicielle et une version matérielle. La première version développe une implémentation logicielle pour des processeurs à usage général. Pour cela, de nouvelles implémentations de l'algorithme tenant compte des ressources mémoire et de l'exécution séquentielle des processeurs ont été proposées. La deuxième version développe de nouvelles architectures de processeurs particularisés pour FPGA. Elles tiennent compte des ressources de calcul disponibles, des ressources mémoire et du potentiel de parallélisation à grain fin offert par le FPGA. De plus, une comparaison avec une version logicielle modifiée est effectuée.

Dans les deux cas, les performances et les compromis pour la sélection de différentes structures de données de nœuds en mémoire ont été analysés. Une sélection de paramètres est proposée afin de maximiser la fonction objective de performance qui combine le nombre de cycles, la consommation mémoire et la fréquence d'horloge du système. Les paramètres permettent de déterminer lequel des deux ou des trois types de structures de données de nœuds (selon la version) sera choisi pour chaque nœud d'une machine à états.

Lors de la validation, des scénarios de test utilisant des données variées ont été utilisés afin de s'assurer du bon fonctionnement de l'algorithme. De plus, les contenus des règles de Snort 2.9.7 ont été utilisés. La machine à états a été construite avec environ $26\times10^3$ chaînes de caractères qui sont toutes extraites de ces règles. La machine à états contient environ $381\times10^3$ nœuds.

La contribution générale de ce mémoire est de montrer qu'il est possible, à travers l'exploration d'architectures, de sélectionner des paramètres afin d'obtenir un produit *mémoire* × *temps* optimal. Pour ce qui est de la version logicielle, la consommation mémoire diminue de 407 Mo à 21 Mo, ce qui correspond à une diminution de mémoire d'environ 20× par rapport au pire cas avec seulement un type de nœud. Pour ce qui est de la version matérielle, la consommation mémoire diminue de 11 Mo à 4 Mo, ce qui résulte en une diminution de mémoire d'environ 3× par rapport à la version logicielle modifiée. Pour ce qui est du débit, il augmente de 300 Mbps pour la version logicielle modifiée à 400 Mbps pour la version matérielle.

# ABSTRACT

This work proposes effective methods and architectures for the implementation of the Aho-Corasick algorithm. This algorithm can be used for pattern matching in network-based intrusion detection systems such as Snort.

Two versions are proposed, a software version and a hardware version. The first version develops a software implementation in C/C++ for general purpose processors. For this, new implementations of the algorithm, considering the memory resources and the processor's sequential execution, are proposed. The second version develops an architecture in VHDL for a specialized processor on FPGA. For this, new architectures of the algorithm, considering the available computing resources, the memory resources and the inherent parallelism of FPGAs, are proposed. Furthermore, a comparison with a modified software version is performed.

For both cases, we analyze the performance and cost trade-off from selecting different data structures of nodes in memory. A selection of parameters is used in order to maximize de performance objective function that combines the cycles count, the memory usage and the system's frequency. The parameters determine which of two or three types of data structures of nodes (depending on the version) is selected for each node of the state machine.

For the validation phase, test cases with diverse data are used in order to ensure that the algorithm operates properly. Furthermore, the Snort 2.9.7 rules are used. The state machine was built with around $26\times10^3$ patterns which are all extracted from these rules. The state machine is comprised of around $381\times10^3$ nodes.

The main contribution of this work is to show that it is possible to choose parameters through architecture exploration, to obtain an optimal *memory* × *time* product. For the software version, the memory consumption is reduced from 407 MB to 21 MB, which results in a memory improvement of about 20× compared with the single node-type case. For the hardware version, the memory consumption is reduced from 11 MB to 4 MB, which results in a memory improvement of about 3× compared with the modified software version. For the throughput, it increases from 300 Mbps with the modified software version to 400 Mbps with the hardware version.

# TABLE DES MATIÈRES

DÉDICACE ..... III

REMERCIEMENTS ..... IV

RÉSUMÉ ..... V

ABSTRACT ..... VI

TABLE DES MATIÈRES ..... VII

LISTE DES TABLEAUX ..... X

LISTE DES FIGURES ..... XI

LISTE DES SIGLES ET ABRÉVIATIONS ..... XIII

CHAPITRE 1 INTRODUCTION ..... 1

1.1 Mise en contexte ..... 1

1.2 Problématique et motivation ..... 2

1.3 Objectifs du mémoire ..... 3

1.4 Méthodologie ..... 4

1.5 Contributions ..... 4

1.6 Organisation du mémoire ..... 5

CHAPITRE 2 LES AUTOMATES FINIS ET LEURS IMPLÉMENTATIONS ..... 6

2.1 Algorithme Aho-Corasick ..... 6

2.1.1 Automates finis et graphes ..... 6

2.1.2 Description ..... 7

2.1.3 Étape 1 de construction ..... 7

2.1.4 Étape 2 de recherche ..... 8

2.2 Revue d’implémentations de l’algorithme Aho-Corasick ..... 9

2.2.1 Représentations des nœuds ..... 9

2.2.2 Architectures de la machine à états ........ 11

2.2.3 Parallélisation des entrées ........ 13

2.2.4 Pipelines ........ 15

2.2.5 Sommaire et sélection de la méthode ........ 16

CHAPITRE 3 STRUCTURES DE DONNÉES POUR UNE IMPLÉMENTATION LOGICIELLE DE L'ALGORITHME AHO-CORASICK ........ 18

3.1 Distribution des nœuds ........ 18

3.2 Structures de données d'un nœud ........ 18

3.2.1 Structure de données d'un nœud de type 1 ........ 19

3.2.2 Structure de données d'un nœud de type 2 ........ 20

3.2.3 Structure de données d'un nœud de type 3 ........ 21

3.3 Distribution des nœuds de type 3 ........ 21

3.4 Construction de l'automate fini ........ 22

CHAPITRE 4 CONCEPTION D'UN PROCESSEUR SPÉCIALISÉ ........ 24

4.1 Types de nœuds modifiés ........ 24

4.1.1 Structure de données d'un nœud de type 1 modifiée (1M) ........ 25

4.1.2 Structure de données d'un nœud de type 2 modifiée (2M) ........ 25

4.2 Architecture matérielle ........ 26

4.2.1 Type 1M uniquement ........ 26

4.2.2 Type 2M uniquement ........ 27

4.2.3 Système complet (types 1M et 2M) ........ 28

CHAPITRE 5 MÉTHODOLOGIE, RÉSULTATS ET DISCUSSIONS ........ 31

5.1 Scénarios de test ........ 31

5.2 Équipement utilisé ........ 31

5.3 Fonctions objectives ........ 32

5.4 Exécution des tests ....................................................................................... 33

5.4.1 Version logicielle ....................................................................................... 33

5.4.2 Version matérielle ....................................................................................... 33

5.5 Résultats pour la version logicielle ....................................................................................... 34

5.6 Résultats pour la version matérielle ....................................................................................... 39

5.7 Discussion ....................................................................................... 44

CHAPITRE 6 CONCLUSION ET TRAVAUX FUTURS ....................................................................................... 48

6.1 Synthèse des travaux et contributions ....................................................................................... 48

6.2 Limitations ....................................................................................... 50

6.3 Travaux futurs ....................................................................................... 51

RÉFÉRENCES ....................................................................................... 53

# LISTE DES TABLEAUX

Tableau 2.1: Exemple de STT (basé sur Tran et al. [24]) ........................................................ 11

Tableau 2.2: États actifs pour l'entrée « scrubya » (basé sur Pao et al. [37]) ............................... 15

Tableau 2.3: Sommaire de la revue de littérature........................................................................... 17

Tableau 5.1: Définition des scénarios de test ................................................................................. 31

Tableau 5.2: Comparaison avec la littérature ................................................................................. 47

# LISTE DES FIGURES

Figure 1.1: Exemple d'une règle de Snort [7]........................................................................2

Figure 2.1: Exemples de machine à états orientée (a), et sous forme d'arbre orienté (b)...............7

Figure 2.2: Exemple d'automate fini ..........................................................................................8

Figure 2.3: Architecture utilisant deux types de représentation (basé sur Shenoy et al. [23]).......10

Figure 2.4: Procédure de création des plusieurs FSM (basé sur Jung et al. [27])..........................11

Figure 2.5: Architecture utilisant des FSM (basé sur Dandass et al. [29]) ..................................12

Figure 2.6: Exemple d'architecture avec CLT (basé sur Piyachon et al. [31])............................13

Figure 2.7: Exemple d'entrées vérifiées en parallèle (basé sur Arudchutha et al. [35])...............14

Figure 3.1: Nombre de nœuds ayant un nombre spécifique de prochains nœuds.........................19

Figure 3.2: Nœud de type 1 .......................................................................................................20

Figure 3.3: Nœud de type 2.......................................................................................................20

Figure 3.4: Nœud de type 3.......................................................................................................21

Figure 3.5: Nombre de nœuds ayant un tableau d'une taille spécifique (nœud de type 3)............22

Figure 4.1: Nœud de type 1M ....................................................................................................25

Figure 4.2: Nœud de type 2M ....................................................................................................26

Figure 4.3: Exemple d'architecture avec des nœuds de type 1M.................................................27

Figure 4.4: Exemple d'architecture avec des nœuds de type 2M.................................................28

Figure 4.5: Architecture générale (types 1M et 2M)...................................................................29

Figure 4.6: Architecture ARCH1_4 ...........................................................................................29

Figure 5.1: Schéma d'exécution des tests (version logicielle) .....................................................33

Figure 5.2: Schéma d'exécution des tests (version matérielle)....................................................34

Figure 5.3: Valeur de *n* ayant la meilleure performance avec les types de nœuds 1, 2 et 3 ..........35

Figure 5.4: Performance lorsque $n = 1$ en utilisant les types de nœuds 1, 2 et 3.........................36

Figure 5.5: Sensibilité à la variation de *m* ..... 36

Figure 5.6: Performance ..... 37

Figure 5.7: Performance avec agrandissement ..... 38

Figure 5.8: Utilisation mémoire ..... 38

Figure 5.9: Nombre de cycles par caractère ..... 39

Figure 5.10: Temps de synthèse ..... 40

Figure 5.11: Nombre de BRAM utilisé avec Xilinx XC7V2000T ..... 40

Figure 5.12: Fréquence de l'horloge ..... 41

Figure 5.13: Nombre de registres et nombre de LUT ..... 42

Figure 5.14: Utilisation mémoire ..... 42

Figure 5.15: Temps de recherche ..... 43

Figure 5.16: Performance ..... 43

Figure 6.1: Exemples de liens avec la version actuelle (a), et avec la proposition future (b) ..... 52

# LISTE DES SIGLES ET ABRÉVIATIONS

| | |
|---|---|
| ASCII | American Standard Code for Information Interchange |
| BRAM | Block RAM |
| CAM | Content-Addressable Memory |
| CLT | CAM-based Lookup Table |
| CTD | Centre de traitements de données |
| DPI | Deep Packet Inspection |
| FPGA | Field-Programmable Gate Array |
| FSM | Finite State Machine |
| Gbps | Gigabit par seconde |
| Go | Gigaoctet ($10^9$ octets) |
| GPU | Graphics Processing Unit |
| HIDS | Host-based Intrusion Detection System |
| IDS | Intrusion Detection System |
| LUT | Look-Up Table |
| MB | Megabyte ($10^6$ bytes) |
| Mbps | Mégabit par seconde |
| MHz | Mégahertz ($10^6$ hertz) |
| Mo | Mégaoctet ($10^6$ octets) |
| NFA | Nondeterministic Finite Automaton |

| | |
|---|---|
| NIDS | Network-based Intrusion Detection System |
| P-AC | Pipelined-Aho-Corasick |
| RAM | Random Access Memory (mémoire vive) |
| ROM | Read-Only Memory (mémoire morte) |
| STT | State Transition Table |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| UTF-8 | Universal Character Set Transformation Format - 8 bits |
| UTF-16 | Universal Character Set Transformation Format - 16 bits |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit |
| XFA | Extended Finite Automata |

# CHAPITRE 1 INTRODUCTION

## 1.1 Mise en contexte

La sécurité informatique est une préoccupation importante lors de l'utilisation d'internet par un usager, que ce soit pour accéder à son compte bancaire, lire ses courriels ou faire des achats. L'accès à ces informations ainsi que les communications via internet doivent être effectués d'une manière sécuritaire. De manière générale, les informations des clients sont stockées à l'intérieur de serveurs qui se trouvent dans des centres de traitements de données (CTD). Il est donc important de bien sécuriser ces CTD contre des cyberattaques.

Afin d'assurer cette sécurité, un des outils possibles est le pare-feu. Ce dernier peut inclure des systèmes de détection et de prévention d'intrusions afin de détecter des virus, des pourriels, des anomalies, etc. [1]. Un système de détection d'intrusion (*Intrusion Detection System* - IDS) permet de détecter des activités suspectes non autorisées sur un système informatique. Il existe deux types d'IDS, au niveau des hôtes (*Host-based Intrusion Detection System* - HIDS) et au niveau des réseaux (*Network-based Intrusion Detection System* - NIDS). Un HIDS utilise les relevés d'accès des systèmes d'exploitation des hôtes comme première source de détection d'intrusion [2]. Un NIDS automatise le processus de détection d'intrusion en surveillant les paquets réseau provenant de l'extérieur de l'ordinateur ou du réseau et en les analysant pour des signes d'attaques ou de sondes. Ces incidents peuvent avoir plusieurs causes comme un logiciel malveillant ou des pirates informatiques qui accèdent illégalement au système via internet [3]. Il y a plusieurs types d'attaques comme des débordements de mémoire tampon (*buffer overflows*), des balayages de ports furtifs (*stealth port scans*), des attaques d'interfaces de passerelles communes (*common gateway interface attacks*), des sondes pour le « server message block », des prises d'empreintes de la pile TCP/IP (*OS or TCP/IP stack fingerprinting attempts*), etc. [4]. Si ces paquets problématiques peuvent être détectés à temps, il est alors possible de les empêcher de rentrer dans le réseau afin d'éviter différentes attaques.

Un exemple de NIDS est Snort. Il s'agit d'un code source ouvert qui permet d'effectuer une analyse de trafic réseau en temps réel [5]. Pour cela, un paquet est reçu, son en-tête est analysé et sa charge utile est vérifiée. Si tout est valide, le paquet continue son chemin, sinon une alerte est lancée et le paquet pourrait être éventuellement détruit. Afin de vérifier le contenu d'un paquet, il est nécessaire

de développer une manière rapide et efficace pour en comparer les données à une collection de patrons ou de règles. Snort utilise de telles règles afin de détecter des intrusions. Les règles ont l'avantage de détecter des attaques au jour 0 (*0-day detection*), c'est-à-dire qu'il est possible de détecter des attaques précédemment inconnues. Pour développer ces règles, il faut avoir une connaissance accrue du fonctionnement de la vulnérabilité. Les règles de Snort sont soumises par la communauté code source ouvert (*open source*) ou par des intégrateurs de Snort directement [6].

Les règles de Snort suivent une syntaxe spécifique [7] dont un exemple est présenté à la Figure 1.1. Pour cet exemple, une alerte sera lancée lorsque le protocole, l'adresse et le port de la source ainsi que l'adresse et le port de la destination seront identiques à ceux présents dans une règle. De plus, il faut aussi que le contenu du paquet, également surnommé la charge utile, soit identique à la partie « content » de la règle. Pour l'exemple précédent, « BAD COMMAND OR FILENAME » indique qu'il y a eu une tentative d'accès erronée. S'il y a une alerte, la partie « msg » de la règle contient des informations supplémentaires sur la possible intrusion.

```
alert tcp !192.168.1.0/24 any -> 192.168.1.0/24 111 \
(content:"BAD COMMAND OR FILENAME”; msg:”ATTACK-RE-
SPONSES command error”;)
```

Figure 1.1: Exemple d'une règle de Snort [7]

Dans ce travail, nous nous sommes limités à la partie « content » des règles pour la détection d'intrusion. De plus, la détection d'intrusion sera faite en analysant les données des paquets entrant dans un réseau et en les comparant à une collection de patrons, telle que celle de Snort [4]. La comparaison entre des chaînes de caractères extraites des paquets et d'une collection de signatures malicieuses est une des méthodes les plus utilisées dans les NIDS [8]. La version 2.9.7 de Snort [6] utilise l'algorithme Aho-Corasick pour cette comparaison. Pour cette raison, nous utiliserons ce même algorithme.

## 1.2 Problématique et motivation

La majorité des données obtenues via l'internet, que ce soit suite à une recherche avec Google ou une lecture du fil d'actualités sur Facebook, sont traitées par des serveurs à l'intérieur de centres de traitement de données (CTD). Ces serveurs reçoivent nos requêtes, consultent de gigantesques bases de données et nous renvoient l'information demandée.

La consommation d'énergie de ces serveurs est de plus en plus importante. Par exemple, le nombre de recherches Google est passé d'environ 18 millions par jour en 2000 à plus de 3.3 milliards par jour en 2012, un facteur de progression de 183 [9]. En 2009, une recherche Google requérait déjà environ 1 kJ d'énergie [10]. On estime qu'en 2010, les centres de traitement de données aux États-Unis consommaient environ 2% de la quantité d'énergie consommée par ce pays [11]. Il apparaît donc important de minimiser la quantité d'énergie utilisée par les serveurs des centres de traitement des données, tout en maintenant leur niveau de performance en termes de requêtes traitées par seconde et en termes de latence de traitement.

Ces serveurs utilisent majoritairement des microprocesseurs à usage général Intel ou AMD, similaires à ceux de nos ordinateurs personnels. Ces microprocesseurs sont généralement conçus avant tout pour maximiser leurs performances. Une alternative aux microprocesseurs à usage général consiste à exploiter le parallélisme à grain fin offert par les réseaux logiques programmables (*Field-Programmable Gate Array* - FPGA) et à concevoir des processeurs sur mesure particularisés pour les tâches uniques des requêtes web. Des compagnies telles que Microsoft étudient d'ailleurs cette possibilité [12]. L'acquisition récente d'Altera par Intel [13] devrait permettre l'introduction de nouveaux produits combinant des processeurs à usage général avec des FPGA pour les CTD. Dans ce mémoire, la question principale est de déterminer s'il est plus performant d'utiliser des FPGA plutôt que des processeurs standards dans des CTD. Une sous-question est de déterminer s'il est possible d'exploiter le parallélisme des FPGA à cet effet.

## 1.3 Objectifs du mémoire

L'objectif général de ce travail est de proposer des méthodes et architectures efficaces pour l'implémentation d'un IDS utilisant l'algorithme Aho-Corasick. Une implémentation logicielle pour des processeurs à usage général ainsi qu'une implémentation matérielle pour un processeur sur mesure dédié à la détection d'intrusions dans les CTD seront développées dans le but d'être le plus performant et sécuritaire possible. Pour cela, nous ciblerons trois objectifs spécifiques :

1. Identifier une application de détection d'intrusions d'un CTD présentement exécutée sur des processeurs à usage général.
2. Proposer des versions logicielles plus efficaces.
3. Proposer des architectures matérielles plus efficaces.

4. Proposer différentes représentations des informations en mémoire.
5. Vérifier le fonctionnement correct de l'application, mesurer leur débit de traitement et consommation de ressources.

## 1.4 Méthodologie

Une implémentation logicielle pour des processeurs à usage général ainsi qu'une implémentation matérielle pour un processeur sur mesure seront développées. Dans un premier temps, il faudra, pour la même application que précédemment, proposer des versions logicielles plus efficaces, qui tiennent compte des ressources mémoire et de l'exécution séquentielle des processeurs à usage général. De plus, il faudra proposer des versions et architectures matérielles plus efficaces, qui tiennent compte des ressources de calcul disponibles et du parallélisme inhérent des FPGA. Ensuite, il faudra développer, pour l'application choisie, différentes représentations des nœuds en mémoire. Pour la version logicielle, ces représentations seront développées en C/C++. Pour ce qui est de la version matérielle, des chemins de données et des unités de contrôle pour des processeurs particularisés seront proposés et décrits dans un langage de description matérielle comme VHDL (VHSIC Hardware Description Language). Cette description sera ensuite synthétisée pour FPGA. Finalement, il faudra vérifier le fonctionnement correct de l'application et des diverses implémentations développées grâce à des bancs d'essai réalistes utilisant des banques de données représentatives pour l'application. Du point de vue performance, il faudra réaliser des mesures telles que le débit de traitement, la fréquence du système et la consommation de ressources.

## 1.5 Contributions

Ce mémoire contient quatre contributions principales.

La première contribution de ce travail est la proposition de trois types de nœuds pour une version logicielle ainsi que deux types de nœuds pour une version matérielle. Le premier type de chaque version sauvegarde les caractères et les pointeurs des prochains nœuds possibles. Le deuxième type de chaque version sauvegarde un pointeur pour chacun des 256 caractères de la table ASCII étendue. Le troisième type sauvegarde un pointeur pour tous les caractères de la table ASCII étendue compris entre les valeurs du plus petit et du plus gros caractère présents dans les prochains nœuds.

La deuxième contribution est une manière de choisir entre ces deux ou trois types. En effet, ce travail montre qu'il est possible de choisir des paramètres afin d'obtenir un produit *mémoire* × *temps* optimal.

La troisième contribution est un article publié au *ACM/IEEE Symposium on Architectures for Networking and Communications Systems 2016* [14]. Cet article montre qu'il est possible de choisir un paramètre afin d'obtenir un produit *mémoire* × *cycles* optimal. Ce produit mène à une amélioration de performance de 12×. Ils proposent une configuration de nœuds parmi les nœuds de type 1 et 2.

La quatrième contribution est la proposition de plusieurs architectures matérielles utilisant deux types de nœuds et permettant d'effectuer la recherche d'une manière efficace en prenant avantage du parallélisme inhérent au FPGA.

## 1.6 Organisation du mémoire

Le reste de ce mémoire est organisé comme suit. Tout d'abord, les automates finis et leurs implémentations seront présentés au Chapitre 2, incluant une revue de littérature. Ensuite, la configuration de nœuds en logiciel (version logicielle) avec un processeur à usage général sera présentée au Chapitre 3 et la conception d'un processeur spécialisé (version matérielle) sera présentée au Chapitre 4. Par la suite, au Chapitre 5, la méthodologie, les résultats et les discussions seront présentés. Finalement, la conclusion et les travaux futurs seront au Chapitre 6.

# CHAPITRE 2 LES AUTOMATES FINIS ET LEURS IMPLÉMENTATIONS

Dans ce chapitre, les automates finis ainsi que leurs implémentations seront présentés. Nous présentons d'abord l'algorithme Aho-Corasick, avec une introduction aux automates finis et aux graphes. Une description de l'algorithme sera ensuite présentée, ainsi qu'un exemple de sa construction et de son fonctionnement lors d'une recherche. Finalement, une revue de littérature avec différentes architectures, méthodes et idées pour améliorer l'algorithme Aho-Corasick ainsi qu'un sommaire de ces derniers, seront présentés.

## 2.1 Algorithme Aho-Corasick

Le sujet de recherche principal de ce mémoire est basé sur l'algorithme Aho-Corasick. Avant de se pencher sur cet algorithme, quelques explications et définitions utiles seront présentées telles que les automates finis et les graphes. Avec ces notions, l'algorithme pourra ensuite être présenté.

### 2.1.1 Automates finis et graphes

Quelques notions sont nécessaires afin de bien comprendre la recherche liée à ce mémoire. La première notion est celle des automates finis ou machines à états finis (*Finite State Machine* - FSM). Un automate fini contient un nombre déterminé (fini) d'états représentés par des nœuds. L'état courant, qui est unique, fait toujours partie des états disponibles de l'automate. Les changements d'état dépendent de l'état courant et de l'entrée courante. Ceci implique qu'une entrée n'aura pas toujours le même effet sur les changements d'état.

Il y a deux types de FSM, la machine de Moore et celle de Mealy. La différence entre les deux est reliée aux sorties de la machine. En effet, la sortie de la machine de Moore dépend uniquement de son état courant tandis que celle de Mealy dépend de son état courant et de l'entrée courante.

Il y a plusieurs représentations possibles de ces machines à états. Une de celles-ci est la représentation avec un graphe. Les nœuds représentants les états ont généralement la forme d'un cercle et les transitions sont représentées par des flèches ou des lignes telles que dans les exemples de la Figure 2.1. Le graphe (b) est une représentation sous la forme d'un arbre avec des branches orientées partant du haut, appelé « nœud racine », jusqu'en bas, appelé « feuilles ». [15]

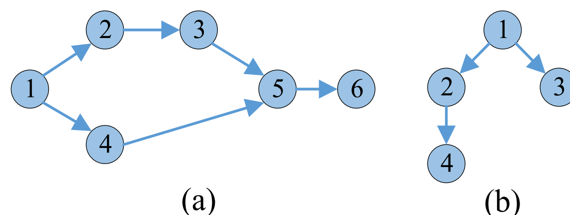


Figure 2.1: Exemples de machine à états orientée (a), et sous forme d'arbre orienté (b)

### 2.1.2 Description

L'algorithme Aho-Corasick décrit par Alfred V. Aho et Margaret J. Corasick en 1975 [16] est utilisé dans la version Snort 2.9.7 [6]. Il utilise une FSM de Moore pour la recherche de chaînes de caractères. Il y a deux types d'algorithmes possibles pour la recherche, le premier étant celui à mots-clés uniques (la recherche va trouver un seul mot commun et arrêter) et le deuxième étant celui à mots-clés multiples (la recherche va trouver tous les mots communs). L'algorithme Aho-Corasick effectue la recherche à mots-clés multiples et peut être divisé en deux étapes distinctes.

### 2.1.3 Étape 1 de construction

La première étape consiste à créer l'automate fini tel que montré à la Figure 2.2. Pour cela, un trie ou arbre préfixe avec des mots d'entrée provenant d'un dictionnaire doit être construit. En effet, il faut créer un automate fini en rajoutant des chaînes de caractères, donc des mots, ou des règles dans notre cas, caractère par caractère, en partant du nœud racine. Par la suite, une fonction d'échec (les lignes courbes dans la Figure 2.2) pour tous les nœuds doit être trouvée afin de pouvoir « tomber » sur un autre nœud dans le cas d'un caractère non présent dans l'arbre. Par exemple, dans la Figure 2.2, si nous commençons par ajouter les règles « duck », « rugby » et « scrub », elles vont toutes partir du nœud racine et se créer une nouvelle branche. Par la suite, en rajoutant la règle « rubber », puisque le trie a déjà une règle qui commence par « ru », une nouvelle branche va être créée après « ru ». Il en est de même pour la règle « ruby » rajoutée à la branche « rubber ». De plus, le dernier caractère de chaque règle apparaît en gras pour signifier qu'une correspondance a été trouvée. Le fait d'utiliser cet algorithme efficace permet d'éviter de toujours retourner au nœud racine à chaque fois qu'un caractère ne se trouve pas dans la liste des prochains nœuds possibles.

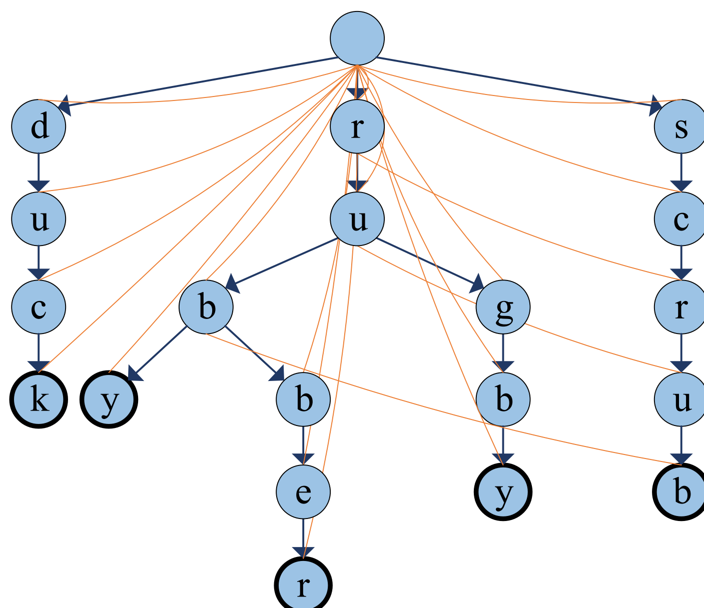


Figure 2.2: Exemple d'automate fini

### 2.1.4 Étape 2 de recherche

La deuxième étape consiste à la recherche de mots-clés en soi. La recherche doit se déplacer à l'intérieur de l'automate dépendamment de l'état présent et d'un caractère lu d'un flux d'entrée. En effet, il faut alimenter un texte d'entrée à l'automate, puis, une recherche en profondeur, caractère par caractère, est effectuée. Si le caractère recherché n'est pas présent dans les prochains nœuds du nœud présent, la fonction d'échec est prise. Par la suite, l'algorithme continue en revérifiant le dernier caractère reçu et poursuit avec le prochain caractère du texte d'entrée. Les mots trouvés à chaque nœud sont extraits directement de l'automate [17]. Par exemple, si l'on recherche le mot « scruby », la recherche va se déplacer dans la branche de droite. En arrivant à la lettre « b », il y aura une correspondance avec le mot « scrub ». Ensuite, l'algorithme va chercher la lettre « y » qui n'est pas présente, c'est pourquoi elle va remonter la ligne courbe jusqu'à l'autre « b » pour ensuite rechercher la lettre « y » et trouver une autre correspondance : « ruby ». Il y a plusieurs implémentations dérivées de l'algorithme Aho-Corasick utilisant différentes structures de données afin de représenter l'automate [14].

## 2.2 Revue d'implémentations de l'algorithme Aho-Corasick

Plusieurs architectures de l'algorithme Aho-Corasick sont trouvées dans la littérature. Les prochaines sous-sections font un résumé des caractéristiques des implémentations de divers auteurs.

### 2.2.1 Représentations des nœuds

Plusieurs représentations des nœuds peuvent être utilisées. Les trois méthodes trouvées dans la littérature sont présentées ci-dessous.

La première méthode est celle de la compression matricielle (*bitmap compression*). Elle est décrite par Tuck et al. [18] qui se basent sur une version non optimisée, de l'algorithme Aho-Corasick où tous les nœuds ont un tableau de 256 pointeurs vers les prochains nœuds. Ils remplacent ensuite ce tableau par un pointeur vers le premier prochain état valide et utilisent un tableau de 256 bits indiquant si le caractère d'entrée est valide ou si le nœud d'échec doit être utilisé. Si le prochain état est valide, son pointeur est calculé en prenant le premier pointeur valide et en y additionnant toutes les valeurs des bits précédents le bit du caractère recherché du tableau. Zha et al. [19] proposent un raffinement de cette implémentation en séparant le tableau en plusieurs niveaux. Par exemple, si le tableau est de 256 bits, le premier niveau pourrait être séparé en 4 blocs de 64 bits. Pour chaque bloc, le deuxième niveau pourrait être séparé en 4 blocs de 16 bits et ainsi de suite. Par la suite, chaque niveau peut avoir plusieurs représentations possibles telles qu'un tableau de bits plus petit, un tableau récapitulatif ou même une table de correspondance (Look-Up Table - LUT). Ils utilisent plusieurs types dépendamment du nombre de prochains nœuds de chaque nœud.

La deuxième méthode est celle de la compression des chemins (*path compression*). Tuck et al. [18] et Zha et al. [19] utilisent cette méthode en plus de la compression matricielle. Ils ont remarqué que la compression matricielle fonctionnait très bien lorsque les nœuds sont proches de la racine, mais qu'il y avait beaucoup de mémoire perdue lorsque les nœuds sont plus loin. Si un certain nombre de nœuds ayant un seul prochain nœud se suivent, ils sont compressés en un seul nœud. Pour ce qui est de Zha et al. [19], leur compression des chemins est utilisée pour toutes les séquences de nœuds ayant aucun ou un prochain nœud. Bremler-Barr et al. [20] proposent trois améliorations : 1) compresser toutes les séquences de nœuds s'ils ont tous un seul prochain nœud et qu'aucun n'est un nœud d'échec, 2) supprimer tous les nœuds feuilles et de les inclure dans l'avant-dernier nœud de la règle, et 3) compresser les pointeurs. Nishimura et al. [21] proposent d'encoder

les chaînes de caractères avec le codage de Huffman. Dimopoulos et al. [22] proposent d'utiliser une mémoire de transposition des caractères pour compresser les caractères en moins de huit bits.

La troisième méthode est d'utiliser plusieurs types de représentation des nœuds dans une FSM. Un exemple décrit par Bremler-Barr et al. [20] propose d’utiliser des tables de correspondance si un nœud a plus de 64 prochains nœuds. Le prochain nœud est accessible en temps constant. Pour les autres cas, ils utilisent un tableau de paires symbole-état. Afin de trouver le prochain nœud, il faut passer linéairement à travers le tableau. Shenoy et al. [23] proposent une méthode semblable où un type de stockage est utilisé pour le nœud racine et un second type est utilisé pour tous les autres nœuds. Un exemple d'architecture est présenté à la Figure 2.3.

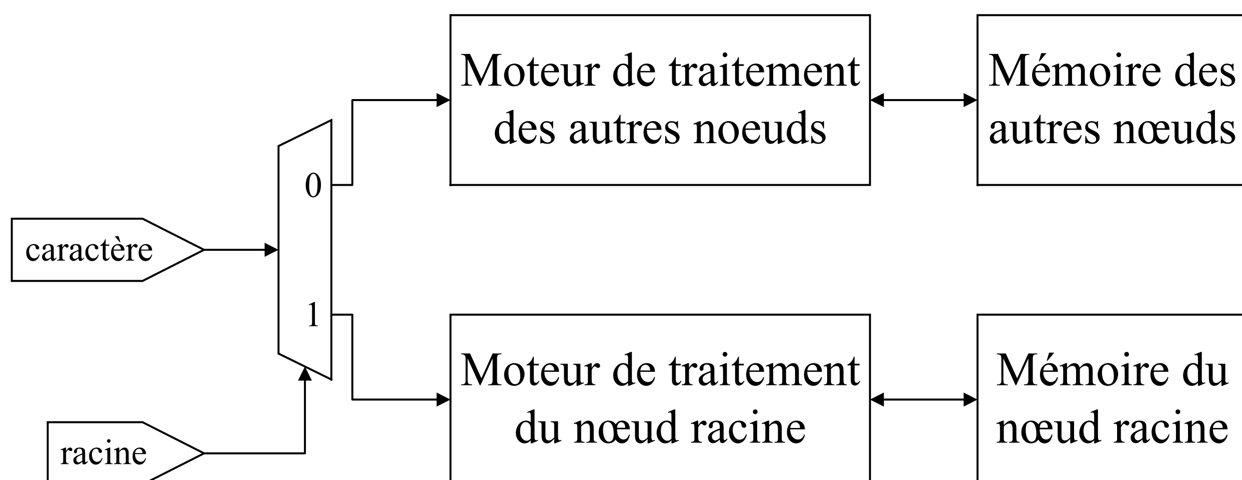


Figure 2.3: Architecture utilisant deux types de représentation (basé sur Shenoy et al. [23])

Dimopoulos et al. [22] proposent aussi d'utiliser deux types de représentation, des tables de correspondances pour les transitions souvent utilisées et des mémoires associatives (*Content-Addressable Memory* – CAM) pour celles moins utilisées. Une représentation semblable à celle de la LUT est présentée par Tran et al. [24]. Afin de limiter le nombre d’accès mémoire, ils organisent leur mémoire avec une table de transitions d’états (*State Transition Table* - STT) qui est un tableau à deux dimensions non modifiable. Ce tableau est sauvegardé dans la mémoire de textures d'un processeur graphique (*Graphics Processing Unit* - GPU) afin qu’elle puisse utiliser la cache de textures qui est sur la puce. Dans une STT, les lignes représentent les prochains états (blanc) d’un état (orange) et les colonnes représentent les caractères d’entrées (vert). Une colonne (bleu) est ajoutée pour savoir s’il y a une correspondance. Un exemple est présenté au Tableau 2.1.

Tableau 2.1: Exemple de STT (basé sur Tran et al. [24])

| | | 0 | 1 | 2 | … | 254 | 255 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 8 |
| 4 | 1 | 0 | 0 | 5 | 0 | 0 | 0 |
| 5 | 0 | 6 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 7 | 0 |

Finalement, une table de hachage sans collisions est proposée par Xu et al. [25]. Cette table de hachage stocke toutes les transitions de la FSM.

### 2.2.2 Architectures de la machine à états

Plusieurs architectures représentant les machines à états peuvent être utilisées. Certaines méthodes trouvées dans la littérature sont présentées ci-dessous.

La première méthode consiste en l'utilisation d'une conjonction de plusieurs petites machines à états. Tan et al. [26] proposent d'utiliser un compilateur de règles qui partitionne et divise (bit-split) une FSM en plusieurs petites tables de transitions. Un exemple de procédure de la création de ces FSM est présenté à la Figure 2.4.

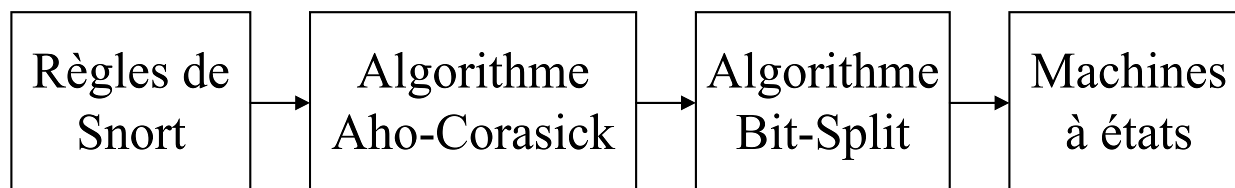


Figure 2.4: Procédure de création des plusieurs FSM (basé sur Jung et al. [27])

Ils partitionnent les règles en plus petits groupes afin de pouvoir combiner les règles ayant des préfixes similaires. Par exemple, pour chaque groupe, on peut diviser la FSM en huit petites FSM, une pour chaque bit du caractère à vérifier. Chaque état a seulement deux prochains états et peut être représenté par un arbre binaire modifié où un nœud peut avoir plusieurs sources à cause des nœuds d’échec. Les huit FSM peuvent être exécutées de manière concurrente. Chaque FSM sauvegarde tous les états traversés dans un tableau et l’intersection des huit tableaux représente l’état

final de sortie. Une idée semblable est proposée par Lin et al. [28]. Jung et al. [27] proposent de rajouter un encodeur de priorité afin de diminuer le nombre de broches d'entrée-sortie. La Figure 2.5 présente un exemple d'une application en bio-informatique.

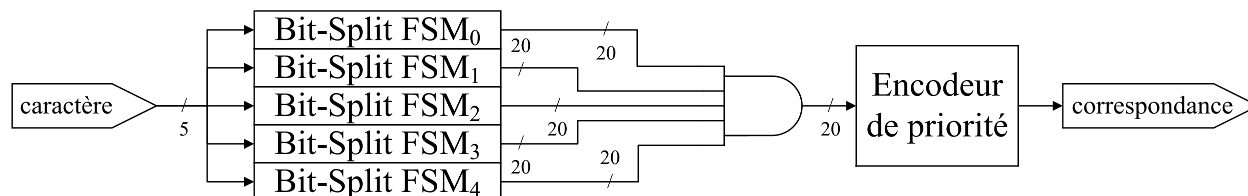


Figure 2.5: Architecture utilisant des FSM (basé sur Dandass et al. [29])

Piyachon et al. [30] proposent trois techniques de partitionnement et de parallélisation des FSM permettant de faire une inspection de paquets en profondeur (*Deep Packet Inspection* - DPI) de haute performance, pour ainsi réduire l'utilisation de la mémoire dans un processeur réseau. La première technique (*bit-level*) consiste à séparer le caractère à rechercher en huit. Ceci implique qu'il y aurait huit recherches effectuées en parallèle dans des FSM. Ceci est semblable au "bit-split" de Tan et al. [26]. La deuxième technique (*byte-level*) compare B octets à la fois. Afin de pouvoir tester toutes les chaînes possibles, il faut exécuter B-1 autres sous-systèmes en parallèle qui commenceraient avec les B-1 autres octets. Par exemple, si l'entrée du système est "ruby", mais que la FSM contient "scruby", il ne trouverait pas de correspondance. Il est important de noter que la mémoire nécessaire reste la même puisque les recherches en parallèle se font dans la même FSM. La troisième technique (*bit-byte-level*) permet de comparer plusieurs bits à la fois. C'est la forme générale des deux techniques précédentes (moins de 8 bits pour bit-level et multiple de 8 bits pour byte-level). Une raison de vouloir remplacer une FSM par plusieurs plus petites FSM est présentée par Lin et al. [28]. En effet, ces petites FSM peuvent possiblement rentrer dans la mémoire partagée du GPU.

Au lieu d'utiliser l'intersection des huit tableaux "bit-split", Piyachon et al. [31] proposent d'utiliser une table de correspondance basée sur une CAM (*CAM-based Lookup Table* – CLT) qui contient les informations des possibles correspondances (la sortie de bit-split). La sortie de cette CAM est une adresse qui pointe vers une table de correspondance qui contient la correspondance trouvée. Un exemple est présenté à la Figure 2.6.

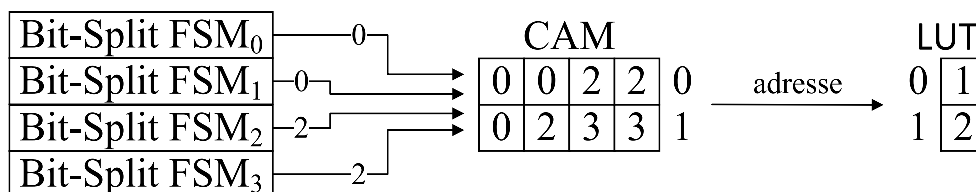


Figure 2.6: Exemple d'architecture avec CLT (basé sur Piyachon et al. [31])

Une autre manière de regrouper les règles en plusieurs FSM est de les classifier selon le type de paquet recherché par la règle. En effet, Dimopoulos et al. [22] proposent de classifier les règles parmi trois à cinq groupes. Chaque groupe est composé de plusieurs petites FSM.

La deuxième méthode consiste à ne pas utiliser des FSM traditionnelles. Par exemple, un automate fini étendu (*Extended Finite Automata* - XFA) est présenté par Smith et al. [32]. Ce XFA est créé en utilisant une FSM traditionnelle et en y rajoutant une mémoire bloc-notes finie (*finite scratch memory*) qui sauvegarde des informations auxiliaires. L'accès et la modification de cette mémoire requièrent l'ajout de nouvelles instructions. Nishimura et al. [21] proposent de numéroter à nouveau les états afin qu'ils soient exprimés en largeur (*breadth-first*) de manière à ce que les nœuds les plus utilisés soient sauvegardés dans la RAM et ensuite dans la cache. Chen et al. [33] proposent de rajouter un arbre suffixe à l'algorithme Aho-Corasick dans les cas où les paquets ne sont pas vérifiés dans le bon ordre. Finalement, Chen et al. [34] proposent d'utiliser un automate fini non déterministe (*Nondeterministic Finite Automaton* - NFA) afin de pouvoir inspecter plusieurs caractères en parallèle. Plusieurs états peuvent donc être activés en même temps. Ils construisent initialement une FSM traditionnelle comme vue précédemment, enlèvent les liens vers les nœuds d'échec et transforment ce FSM en un NFA.

### 2.2.3 Parallélisation des entrées

Plusieurs architectures permettant la parallélisation des entrées peuvent être utilisées. Une méthode trouvée dans la littérature est présentée ci-dessous.

La méthode consiste à séparer l'entrée en groupes de différentes tailles. Le cas où le groupe contiendrait un seul caractère est présenté par Arudchutha et al. [35]. Ils proposent de ne plus avoir de liens vers des nœuds d'échecs, ce que Lin et al. [28] proposent également. De plus, ils proposent de vérifier tous les caractères de l'entrée en parallèle. En effet, chaque fil d'exécution (*thread*) commence à vérifier la chaîne de caractères d'entrée à la position du caractère courant et se termine

lorsque le prochain caractère recherché n'est pas présent dans le nœud courant. La même FSM est donc parcourue en parallèle par tous les fils d'exécutions. Un exemple est présenté à la Figure 2.7.

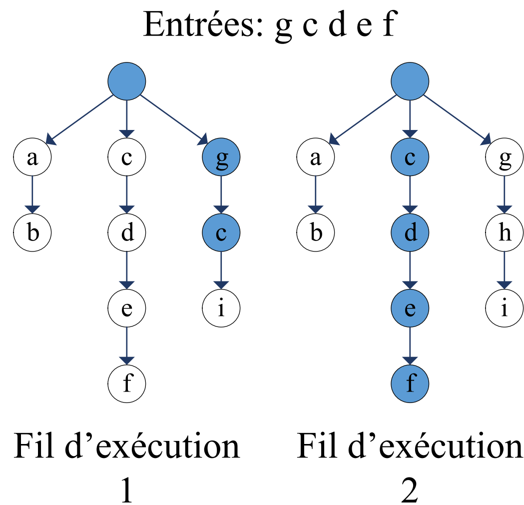


Figure 2.7: Exemple d'entrées vérifiées en parallèle (basé sur Arudchutha et al. [35])

Un groupement plus général est présenté par Tran et al. [24]. Les données d'entrée qui sont dans une mémoire globale sont séparées en fragments de différentes longueurs et envoyées sur plusieurs fils d'exécutions où la recherche est alors effectuée. Afin d'éviter le cas où une chaîne de caractères présente dans les règles est séparée à travers plusieurs fragments, chaque fil d'exécution aura X caractères de plus (où X représente le nombre de caractères de la plus longue règle). Grâce à ces X caractères de plus, le fil d'exécution pourra faire la comparaison complète. Tous ces fils d'exécutions sont exécutés en parallèle sur un GPU. Par la suite, ils proposent de mettre toutes les données d'entrées dans une mémoire partagée qui se trouve sur le GPU. Dans le cas où les entrées sont des paquets TCP/IP, une autre possibilité est d'envoyer un paquet au complet à chaque fil d'exécution du GPU, tel que proposé par Tumeo et al. [36]. Ils proposent aussi d'utiliser différentes mémoires tampons pour accélérer le tout. Une autre manière de regrouper les règles est de les classifier selon le type de paquet recherché par la règle. En effet, Dimopoulos et al. [22] proposent de classifier les règles parmi trois à cinq groupes. Chaque groupe est composé de plusieurs petites FSM qui sont utilisées en parallèle.

### 2.2.4 Pipelines

Plusieurs types de pipelines et d'implémentations matérielles pour l'algorithme Aho-Corasick peuvent être utilisés. Quelques exemples sont présentés ci-dessous.

Le premier exemple de pipeline est proposé par Shenoy et al. [23]. Ils proposent de pipeliner le processus d’accès consécutifs au nœud racine afin de ne pas devoir toujours relire toutes les informations à chaque fois. Une implémentation matérielle pipelinée de l'algorithme Aho-Corasick (P-AC) est présentée par Pao et al. [37]. Toute la recherche est faite avec un pipeline. Chaque niveau de pipeline de leur architecture permet d’effectuer une trace à travers l’automate commençant avec le caractère courant à chaque cycle.

Un exemple d'exécution avec l’entrée « scrubya » utilisant la FSM de la Figure 2.2 est présenté au Tableau 2.2. À chaque cycle, le caractère d’entrée est envoyé à tous les niveaux du pipeline et est utilisé si la nouvelle chaîne existe dans les règles. Le cas échéant, le niveau de pipeline en question est réinitialisé. L’étage 0 est toujours représenté par le nœud racine. Une correspondance peut être trouvée à n’importe quel niveau du pipeline. Dans le cas où la chaîne de caractère d’une règle est plus longue que le nombre de niveaux, la chaîne est séparée en plusieurs segments. Chaque segment contient un identifiant unique et un bit représentant s’il fait partie d’une plus longue chaîne de caractère.

Tableau 2.2: États actifs pour l’entrée « scrubya » (basé sur Pao et al. [37])

| | | État actif des étages du pipeline | | | | | |
|---|---|---|---|---|---|---|---|
| cycle | entrée | étage 0 | étage 1 | étage 2 | étage 3 | étage 4 | étage 5 |
| 1 | s | <racine> | | | | | |
| 2 | c | <racine> | <s> | | | | |
| 3 | r | <racine> | | <sc> | | | |
| 4 | u | <racine> | <r> | | <scr> | | |
| 5 | b | <racine> | | <ru> | | <scru> | |
| 6 | y | <racine> | | | <rub> | | **<scrub>** |
| 7 | a | <racine> | | | | **<ruby>** | |

Dimopoulos et al. [22] proposent aussi de pipeliner la recherche. Leur pipeline contient trois étages. Le caractère d’entrée passe par la mémoire de transposition des caractères (parce que la FSM contient des caractères compresser), puis par les tables de correspondances et les CAM puis finalement, par un module choisissant laquelle des deux mémoires contient le prochain état valide. De plus, c’est dans ce dernier étage qu’un élément sortira du système si une correspondance est trouvée.

### 2.2.5 Sommaire et sélection de la méthode

Un sommaire de la revue de littérature est présenté dans le Tableau 2.3. Les deux premières colonnes présentent le regroupement des architectures et méthodes utilisées par les auteurs. La dernière colonne présente les propositions et les idées de tous les auteurs.

Dans ce mémoire, l'utilisation de plusieurs représentations de nœuds dépendamment d'un seuil, telle que présentée par Bremler-Barr et al. [20] et Shenoy et al. [23] sera exploitée pour la configuration des nœuds. L'utilisation de plusieurs représentations de nœuds permet d'avoir une implémentation proche de la version originale tout en essayant de diminuer la consommation mémoire. Par contre, il faudra que ces changements ne diminuent pas trop le débit du système. Cette configuration des nœuds sera présentée dans les prochaines sections.

Tableau 2.3: Sommaire de la revue de littérature

| Caractéristiques | Méthodes | Publication | Idées |
|---|---|---|---|
| Représentations des nœuds | Compression matricielle (bitmap) | Tuck, 2004 [18] | Utiliser un bitmap |
| | | Zha, 2008 [19] | Utiliser un bitmap à plusieurs niveaux |
| | Compression des chemins | Tuck, 2004 [18] | Compresser des séquences de noeuds en un seul noeud |
| | | Zha, 2008 [19] | Compresser des séquences de nœuds ayant 0 ou 1 prochain nœud en un seul noeud |
| | | Bremler-Barr, 2011 [20] | 1. Compresser toutes les séquences de nœuds s'ils ont tous un seul prochain nœud et qu'aucun est un nœud d'échec<br>2. Supprimer tous les nœuds feuilles et les inclure dans l'avant-dernier nœud de la règle<br>3. Compresser les pointeurs |
| | | Nishimura, 2001 [21] | Encoder les chaînes de caractères avec le codage de Huffman |
| | | Dimopoulos, 2007 [22] | Utiliser une mémoire de transposition des caractères pour compresser les caractères en moins de huit bits |
| | Plusieurs types de représentations des noeuds | Bremler-Barr, 2011 [20] | 1. Utiliser des LUT si un nœud a plus de 64 prochains nœuds<br>2. Utiliser un tableau de pairs symbole-état |
| | | Shenoy, 2011 [23] | 1. Un type de stockage est utilisé pour le nœud racine<br>2. Un second type est utilisé pour tous les autres nœuds |
| | | Dimopoulos, 2007 [22] | 1. Utiliser des tables de correspondances pour les transitions souvent utilisées<br>2. Utiliser des CAM pour les transitions moins utilisées |
| | | Tran, 2012 [24] | Utiliser une STT (semblable à LUT) |
| | | Xu, 2011 [25] | Utiliser des tables de hachage sans collisions |
| Architectures de la machine à états | Conjonction de plusieurs petites machines à états | Tan, 2005 [26] | Bit-split: exemple, diviser la FSM en huit petites FSM |
| | | Jung, 2006 [27] | Rajouter un encodeur à priorité (en plus de bit-split) |
| | | Lin, 2010 [28] | Comme bit-split |
| | | Piyachon, 2006 [30] | Utiliser bit-level, byte-level et bit-byte-level |
| | | Lin, 2010 [28] | Plusieurs petites FSM peuvent possiblement rentrer dans la mémoire partagée du GPU |
| | | Piyachon , 2008 [31] | Utiliser des CLT (en plus d'une variante de bit-split) |
| | | Dimopoulos, 2007 [22] | Utiliser plusieurs FSM dont les règles sont regroupées selon le type de paquet recherché par la règle |
| | Ne pas utiliser des FSM traditionnelles | Smith, 2008 [32] | Utiliser un XFA |
| | | Nishimura, 2001 [21] | Renuméroter les états afin qu'ils soient exprimés en largeur (breadth-first) |
| | | Chen, 2011 [33] | Rajouter un arbre suffixe |
| | | Chen, 2013 [34] | Utiliser un NFA |
| Parallélisation des entrées | Séparer l'entrée en groupes de différentes tailles | Arudchutha, 2013 [35] | 1. Ne plus avoir de liens vers les noeuds d'échec<br>2. Séparer l'entrée caractère par caractère et les vérifier en parallèle |
| | | Lin, 2010 [28] | Ne plus avoir de liens vers les noeuds d'échec |
| | | Tran, 2012 [24] | Séparer l'entrée en fragments de différentes longueurs et les vérifier en parallèle |
| | | Tumeo, 2010 [36] | Vérifier des paquets TCP/IP complets en parallèle |
| | | Dimopoulos, 2007 [22] | Regrouper les règles selon le type de paquet recherché par la règle et exécuter chaque regroupement en parallèle |
| Pipelines | Pipelines | Shenoy, 2011 [23] | Pipeliner le processus d'accès consécutifs au noeud racine |
| | | Pao, 2008 [37] | Implémentation matérielle pipelinée (P-AC) de l'algorithme |
| | | Dimopoulos, 2007 [22] | Pipeliner la recherche en tant que telle |

# CHAPITRE 3 STRUCTURES DE DONNÉES POUR UNE IMPLÉMENTATION LOGICIELLE DE L'ALGORITHME AHO-CORASICK

Suite à la revue de littérature présentée au Chapitre 2, le Chapitre 3 présente trois structures de données de nœuds dans le but de réaliser une implémentation de l'algorithme Aho-Corasick. Tout d'abord, nous présenterons la distribution des nœuds de l'arbre, puis trois structures de nœuds seront proposées et expliquées. Par la suite, la distribution des nœuds du troisième type de structure de données sera présentée. Finalement, la procédure de la construction de l'automate fini sera formulée.

## 3.1 Distribution des nœuds

Avant d'aller plus loin dans la présentation des différentes structures de données de nœuds, il faut d'abord mieux les caractériser à l'aide de profilages à l'exécution. Un automate fini a donc été construit avec 26328 chaînes de caractères qui sont toutes uniques (sans doublons) et extraites des 31133 règles de Snort 2.9.7 [6]. La différence entre les chaînes de caractères et le nombre de règles représente les doublons. Toutes les chaînes de caractères sélectionnées se traduisent en un arbre comprenant 381302 nœuds.

Soit deux nœuds $n_i$ et $n_j$ partageant une transition (arc dirigé). Dans ce qui suit, on nomme $n_j$ le prochain nœud. La Figure 3.1 montre la distribution du nombre de prochains nœuds pour chaque nœud de l'automate fini précédent. Nous remarquons qu'environ 92% des nœuds ont un seul prochain nœud. De plus, lorsque le nombre de prochains nœuds augmente de 0 à 20, le nombre de nœuds diminue de $10^6$ à $10^1$. Par la suite, le nombre de nœuds se stabilise entre 1 et 10. Le nœud ayant 254 prochains nœuds est le nœud racine. En s'inspirant de ces observations, on peut proposer des représentations différentes pour les nœuds selon leur nombre de prochains nœuds.

## 3.2 Structures de données d'un nœud

Tel que présenté à la section 2.1.1, un automate fini est composé de nœuds, représentant des états, et de transitions entre ces états. Dans l'implémentation logicielle choisie, inspirée de Kanani [38], chaque nœud est composé de neuf éléments :

- un entier qui permet de déterminer si le nœud en question contient une correspondance (qu'il représente le dernier nœud d'une des règles de Snort),

- un entier qui représente le nombre de niveaux se trouvant entre le nœud en question et le nœud racine,
- un pointeur vers le nœud d'échec,
- un pointeur vers un tableau contenant toutes les correspondances,
- un entier représentant le nombre de correspondances,
- un entier représentant la capacité maximale dans le tableau de correspondances,
- un entier représentant le type du nœud présent,
- un entier correspondant au nombre de prochains nœuds, et
- un pointeur vers un tableau ou une structure contenant des transitions vers les prochains nœuds.

Ce dernier pointeur peut pointer vers trois différentes structures de données de nœuds. Les sections 3.2.1 à 3.2.3 détaillent ces différentes structures.

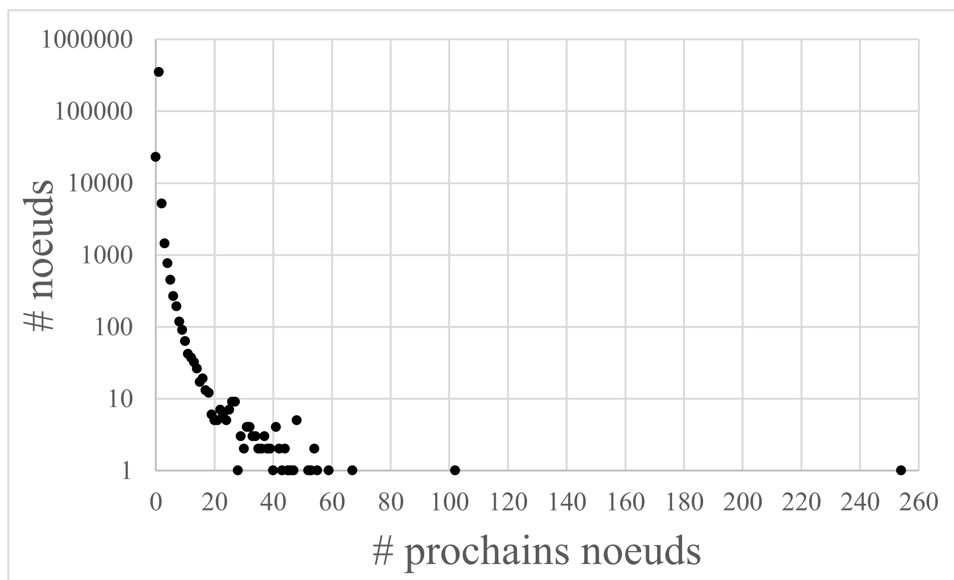


Figure 3.1: Nombre de nœuds ayant un nombre spécifique de prochains nœuds

### 3.2.1 Structure de données d'un nœud de type 1

La première structure de données qu'on nomme *type 1* est présentée à la Figure 3.2. Le type 1 est similaire au tableau de paires symbole-état de Bremler-Barr et al. [20]. Il est composé d'un tableau

contenant *r* structures de deux éléments. Le premier élément contient le caractère à comparer afin de se rendre au prochain nœud. Le deuxième élément est un pointeur vers le prochain nœud correspondant. Pour ce type de nœud, le nombre de structures *r* est égal au nombre de prochains nœuds du nœud en question, ce qui le rend compact. Lors de la recherche, il faut passer à travers toutes les structures du tableau jusqu'à ce qu'une égalité avec le caractère soit trouvée ou que tout le tableau ait été parcouru. La recherche se fait donc dans un temps linéaire dans le pire cas. Il est aussi possible d'utiliser la recherche binaire à travers le tableau, mais cette approche requiert plus de calculs et les avantages, le cas échéant, dépendraient de la taille du tableau. Cette version modifiée n'est pas inclue dans ce mémoire puisque les tailles du tableau sont petites.

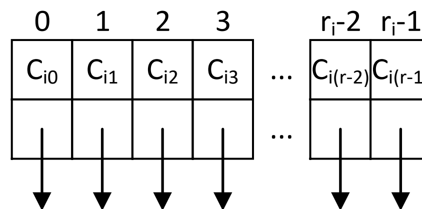


Figure 3.2: Nœud de type 1

### 3.2.2 Structure de données d'un nœud de type 2

La seconde structure de données de *type 2* est présentée à la Figure 3.3. Le type 2 est similaire aux tableaux de correspondance de Bremler-Barr et al. [20]. Il est composé d'un tableau contenant des pointeurs, un pour chacun des 256 caractères de la table ASCII étendue ou UTF-8. La valeur ASCII de chaque caractère est utilisée comme index pour accéder à leur pointeur respectif, s'il existe. La recherche se fait donc en temps constant. Si un index n'est pas associé à un pointeur valide, ce dernier aura la valeur NULL. Ce type de nœud peut mener à du gaspillage de mémoire.

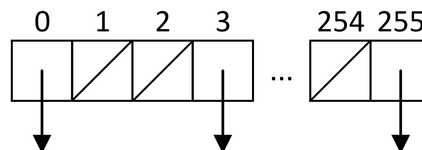


Figure 3.3: Nœud de type 2

### 3.2.3 Structure de données d'un nœud de type 3

Finalement, la troisième structure de données de *type 3* est présentée à la Figure 3.4. Le type 3 est composé d'un tableau contenant des pointeurs, d'un caractère ($C_{i0}$) avec la valeur ASCII la plus petite qui est présent dans le tableau et d'un nombre ($s_i$) qui représente le nombre d'éléments dans le tableau. Le tableau est similaire à celui du type 2 mais contient uniquement les pointeurs des caractères entre $C_{i0}$ et $C_{i0} + s_i - 1$. Ces deux caractères représentent le caractère ayant la valeur ASCII la plus petite et celui ayant la plus grande. Par exemple, si le tableau de type 2 contient seulement des pointeurs valides entre les caractères « a » et « z », $C_{i0}$ serait égal à « a », $s_i$ serait égal à 26 et le tableau contiendrait donc 26 pointeurs. Cette fois-ci, l'index du tableau correspond à la valeur d'entrée recherchée moins la valeur ASCII du caractère $C_{i0}$. Si cette valeur est plus petite que 0 ou plus grande que $s_i$ - 1, on considère que le pointeur est NULL, sinon on accède au pointeur désiré. Dans l'exemple précédent, si l'on recherche le caractère « c », on fera le calcul « c » - « a » qui sera égal à 2 et le prochain pointeur sera donc celui égal à l'index 2. En comparant avec le nœud de type 2, nous remarquons que le gaspillage de mémoire est diminué. La recherche se fait encore en temps constant.

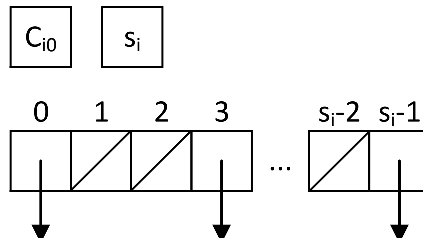


Figure 3.4: Nœud de type 3

## 3.3 Distribution des nœuds de type 3

Suite à la distribution des nœuds de la section 3.1 et la présentation des trois types de nœuds de la section 3.2, il faut d'abord déterminer à partir de combien de prochains nœuds doit-on utiliser un nœud de type 2 ou 3 plutôt qu'un nœud de type 1. Pour ceci, il nous faut aussi connaître le type d'optimisation désiré au niveau performance : souhaite-t-on minimiser l'espace mémoire total utilisé (favorisant ainsi les nœuds de type 1) ou le temps de recherche (favorisant les nœuds de type 2 et 3)? Par exemple, le nœud racine devrait logiquement être du type 2 ou du type 3 puisque c'est le nœud occupant le plus d'espace et le plus emprunté.

La Figure 3.5 montre la distribution de la taille du tableau des nœuds de type 3, considérant que seulement ces derniers sont utilisés. Lorsque le nombre de prochains nœuds augmente de 0 à 100, le nombre de nœuds diminue de $10^6$ à $10^1$. Par la suite, le nombre de nœuds de stabilise entre 1 et 10.

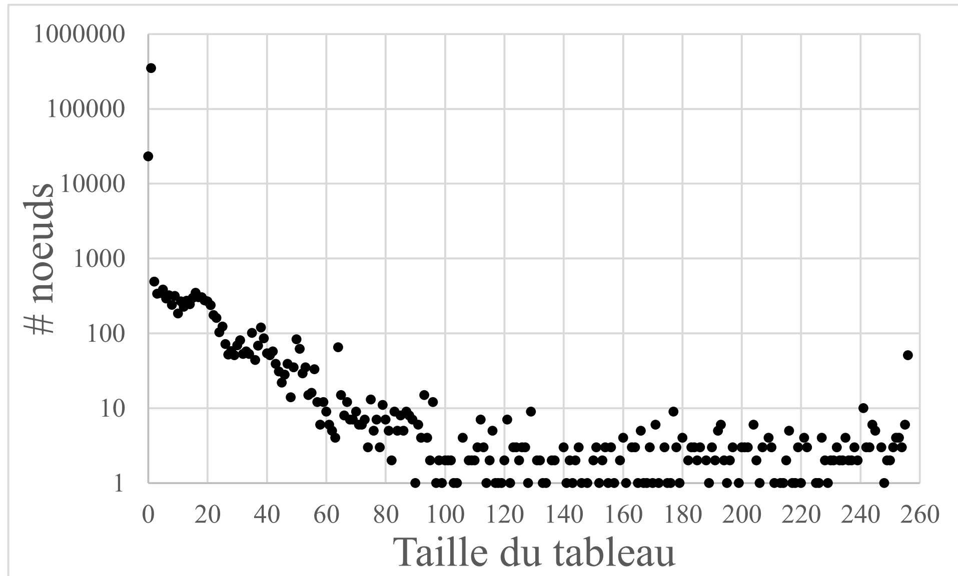


Figure 3.5: Nombre de nœuds ayant un tableau d'une taille spécifique (nœud de type 3)

Suite aux observations précédentes et la distribution des nœuds de type 3, il faut trouver à partir de quelle taille du tableau de prochains nœuds devrait-on utilisé le nœud de type 2 au lieu de celui du type 3. Pour ceci, il faut à nouveau déterminer l'optimisation désirée, soit la minimisation de l'espace mémoire totale utilisée (favorisant le nœud de type 3) ou le temps pour effectuer la recherche (favorisant le nœud de type 2). Pour le même exemple que précédemment, le nœud racine devrait logiquement être du type 2 puisque c'est le nœud occupant le plus d'espace et le plus visité.

Considérant les deux distributions de ce chapitre ainsi que la présentation des trois types de nœuds proposés, la section suivante présentera la méthode proposée pour la construction de l'automate fini.

## 3.4 Construction de l'automate fini

Lors des expériences, un automate fini de l'algorithme Aho-Corasick a été construit comme suit. Tous les nœuds sont initialement construits avec le nœud de type 2 comprenant un tableau de taille 256. Après l'inspection de tous les nœuds, ceux ayant moins ou autant de prochains nœuds $r_i$ qu'un

seuil prédéterminé $n$ ($r_i \leq n$) sont convertis en nœuds de type 1. Par exemple, si le seuil est à $n = 2$, tous les nœuds avec zéro prochain nœud sont convertis en nœuds de type 1 avec aucune structure de deux éléments. Les nœuds ayant un prochain nœud sont convertis en nœuds de type 1 avec une structure de deux éléments. Les nœuds ayant deux prochains nœuds sont convertis en nœuds de type 1 avec deux structures de deux éléments. Si les nœuds ont plus de deux prochains nœuds, ils sont du type 2 ou du type 3.

Pour ce dernier cas, les nœuds qui ont la taille du tableau de type 3 ($s_i$) plus petit ou égal à un second seuil $m$ ($s_i \leq m$), sont convertis en nœuds de type 3. Pour le même exemple que précédemment, quand les nœuds ont plus de deux prochains nœuds et que le seuil est à $m = 3$, pour qu'un nœud soit du type 3, il faut que la taille du tableau de type 3 soit de trois. En conséquence, si la taille du tableau est plus grande que trois, le nœud est du type 2 avec un tableau de 256 éléments. Si un nœud a seulement six prochains nœuds possibles, ce nœud de type 2 a 250 pointeurs NULL dans son tableau.

Deux cas particuliers doivent être considérés. Tout d'abord, lorsque $n = -1$ et $m = -1$, tous les nœuds sont du type 2. Puis, lorsque $n = 256$, tous les nœuds sont du type 1.

Dans ce chapitre, nous avons présenté deux distributions de nœuds, trois structures de données de nœuds, ainsi que la procédure utilisée pour la construction de l'automate fini. Dans le prochain chapitre, il sera question de la conception d'un processeur spécialisé avec ses différentes structures de données de nœuds et ses architectures proposées.

# CHAPITRE 4 CONCEPTION D'UN PROCESSEUR SPÉCIALISÉ

Au chapitre précédent, trois types de structure de nœuds ont été présentés et expliqués. Alors que dans le Chapitre 3, ces structures étaient destinées à un processeur à usage général (implémentation logicielle), dans ce chapitre-ci nous nous concentrons sur la conception de processeurs spécialisés. Pour cela, certaines adaptations sont faites aux structures du Chapitre 3 afin de mieux s'adapter aux architectures de ces processeurs. Ce chapitre débute avec la présentation des deux structures de données modifiées (type 1M et type 2M) qui sont proposées. Par la suite, trois architectures matérielles sont proposées, une utilisant uniquement des nœuds de type 1M, une autre utilisant uniquement ceux de type 2M et finalement, une dernière complète avec un mélange des deux types de nœuds.

## 4.1 Types de nœuds modifiés

Tel que présenté à la section 3.2, dans l'implémentation logicielle, les nœuds étaient composés de neuf éléments. Ces éléments pouvaient jouer trois rôles : 1) ajouter des informations pertinentes pour la recherche, 2) aider la création de l'automate fini ou 3) donner des informations supplémentaires telles que des copies de toutes les correspondances du nœud en question. Lors d'une implémentation matérielle, le concepteur peut avoir le contrôle de la représentation mémoire sur un FPGA. Il a donc été décidé que les éléments inclus dans les nœuds de la version matérielle seront simplifiés par rapport à la version logicielle originale en gardant seulement les informations pertinentes à la recherche. Afin de pouvoir comparer cette nouvelle version matérielle, la version logicielle vue précédemment ne peut être utilisée puisqu'un nœud ne contient pas du tout les mêmes éléments. Une nouvelle version logicielle simplifiée a donc été développée afin de pouvoir les comparer. Les éléments restants d'un nœud dans la version logicielle modifiée sont :

- un entier représentant le nombre de correspondances,
- une structure contenant un pointeur vers le nœud d'échec et son type correspondant,
- un entier représentant le type du nœud présent, et
- une structure contenant un pointeur vers un tableau. Ce tableau contient les transitions vers tous les prochains nœuds ainsi que le type du prochain nœud correspondant. Pour un des types de nœud, il contient aussi un tableau de caractères à correspondre.

Dans ce qui suit, nous proposons deux types de nœuds modifiés (1M et 2M). Ils sont présentés dans les deux prochaines sections.

### 4.1.1 Structure de données d'un nœud de type 1 modifiée (1M)

La première structure de données modifiée pour FPGA qu'on nomme *type 1M*, est basée sur le nœud de type 1 vu précédemment et est présentée à la Figure 4.1. Le type 1M est composé de trois tableaux contenant *r* éléments. Le premier tableau contient les caractères à comparer afin de se rendre au prochain nœud. Le deuxième tableau contient les types des prochains nœuds correspondants. Finalement, le troisième tableau contient les pointeurs vers le prochain nœud correspondant. Pour ce type de nœud, le nombre d'éléments *r* est égal au nombre de prochains nœuds du nœud en question, ce qui le rend compact. Lors de la recherche, il faut passer à travers tous les éléments des trois tableaux en parallèle. En d'autres mots, l'adresse zéro ainsi que tous les autres indexes de chaque tableau sont lus en même temps. La recherche se fait donc en temps constant.

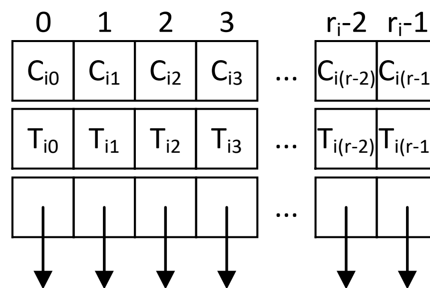


Figure 4.1: Nœud de type 1M

### 4.1.2 Structure de données d'un nœud de type 2 modifiée (2M)

La deuxième structure de données modifiée pour FPGA qu'on nomme *type 2M*, est basée sur le nœud de type 2 vu précédemment et est présentée à la Figure 4.2. Le type 2M est composé de deux tableaux contenant 256 éléments. Le premier contient les types des prochains nœuds et le deuxième contient leur pointeur correspondant. Dans ce cas, les éléments des tableaux sont associés à chacun des 256 caractères de la table ASCII étendue ou UTF-8. La valeur ASCII de chaque caractère est utilisée comme index pour accéder à leur type et leur pointeur respectif, s'ils existent. La recherche se fait donc en temps constant. Si un index n'est pas associé à un type et un pointeur valide, ces

derniers auront respectivement la valeur 0 et la valeur NULL. Ce type de nœud peut mener à du gaspillage de mémoire.

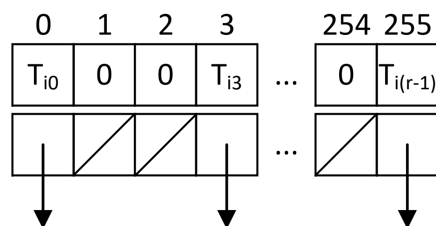


Figure 4.2: Nœud de type 2M

## 4.2 Architecture matérielle

Suite à la présentation des deux types de nœuds proposés, il faut développer une architecture matérielle générale qui prend en compte les deux types de nœuds proposés. L'architecture matérielle comprend un chemin des données et une unité de contrôle. Des exemples d'architectures seront présentés dans les trois sections suivantes. Les deux premières architectures sont des cas spéciaux simplifiés représentant uniquement l'architecture associée au type en question. En effet, le premier cas utilise des nœuds de type 1M et le deuxième cas utilise des nœuds de type 2M. Ces deux cas spéciaux permettent de comprendre le fonctionnement de l'architecture de chaque type.

### 4.2.1 Type 1M uniquement

La première architecture, montrée à la Figure 4.3, présente le cas où tous les nœuds sont du type 1M avec quatre prochains nœuds. Le bloc ROM1_4 représente la mémoire morte (*Read-Only Memory* - ROM) contenant les informations des nœuds. Ce type de mémoire permet seulement la lecture en mémoire. Une ROM est utilisée puisqu'il n'est pas nécessaire d'écrire dans la mémoire après l'initialisation. L'entrée du bloc représente l'adresse à laquelle la ROM sera accédée. Cette adresse est représentée avec un bus contenant 19 bits. Le nombre de bits dépend du nombre maximal d'adresses possibles. Les sorties du bloc sont séparées en cinq groupes. Les quatre premiers vont de 0 à 3 et sont similaires puisqu'ils contiennent l'adresse et le caractère des quatre prochains nœuds. Le caractère est représenté par un bus de 8 bits et contient des valeurs correspondant à la table ASCII ou UTF-8. Pour ce qui est de l'adresse, elle est représentée par un autre bus de 19 bits. Le cinquième groupe contient uniquement l'adresse d'échec et est aussi représenté par un bus de

19 bits. Les caractères des quatre premiers groupes sont comparés avec un caractère d'entrée, aussi sur 8 bits. Les quatre comparaisons se font toutes en parallèle et retournent chacune un bit représentant si une égalité a été trouvée. Ces résultats sont ensuite envoyés à un encodeur. La première sortie de cet encodeur, représentée par un bit, retourne à l'extérieur du système si l'adresse d'échec a été utilisée ou non. La deuxième sortie de l'encodeur est un bus de trois bits et permet de sélectionner parmi les cinq adresses d'entrées du multiplexeur provenant des cinq groupes de la ROM. La sortie du multiplexeur correspond à l'adresse sélectionnée et représente donc la prochaine adresse à vérifier. Cette adresse est encore une fois représentée par un bus de 19 bits et est stockée dans un registre de 19 bits pour être ensuite renvoyée à la ROM. Le caractère d'entrée provient soit de l'extérieur du système, soit d'un registre de huit bits stockant le caractère précédemment utilisé. Ce choix est effectué grâce à un multiplexeur dont le contrôle est effectué grâce au bit de sortie de l'encodeur, qui représente si un nœud d'échec a été utilisé ou non.

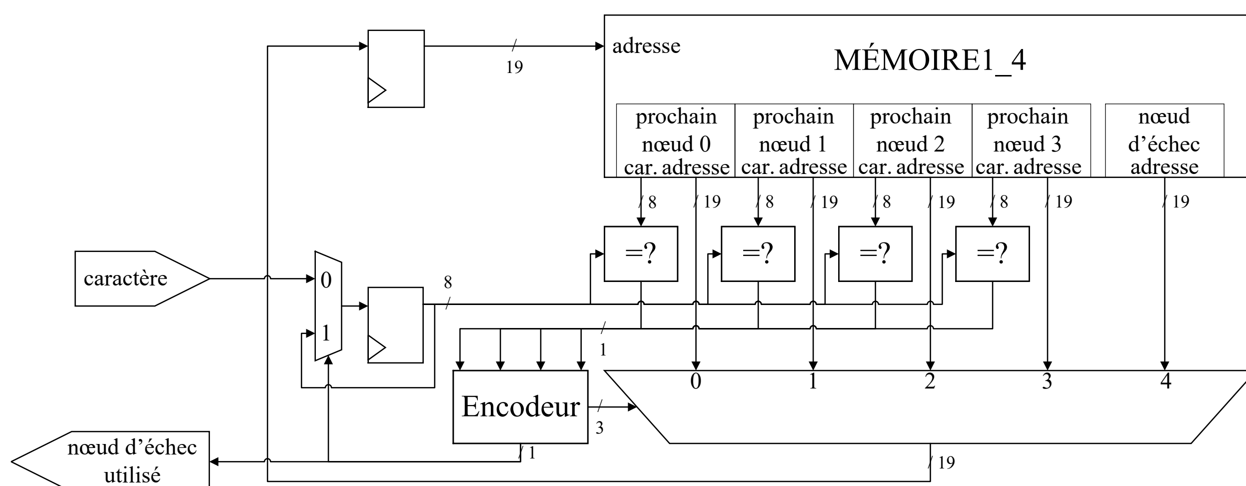


Figure 4.3: Exemple d'architecture avec des nœuds de type 1M

### 4.2.2 Type 2M uniquement

La deuxième architecture, montrée à la Figure 4.4, présente le cas où tous les nœuds sont du type 2M. Le bloc ROM2 représente la ROM contenant les informations des nœuds. Encore une fois, une ROM est utilisée puisqu'il n'est pas nécessaire d'écrire dans la mémoire après l'initialisation. L'entrée du bloc représente l'adresse à laquelle la ROM sera accédée. Cette adresse est représentée avec un bus contenant 27 bits. Grâce au module « calcul d'adresse », cette adresse est calculée avec une adresse de 19 bits qui dépend du nombre maximal d'adresses possibles (tel que présenté à la

section précédente). L'adresse de 19 bits est ensuite concaténée avec le caractère d'entrée de huit bits, ce qui donne une nouvelle adresse de 27 bits. Les sorties du bloc sont l'adresse du prochain nœud ainsi que l'adresse d'échec. Chacune de ces deux dernières est représentée par un bus de 19 bits. L'adresse du prochain nœud est envoyée dans un encodeur et ce dernier a une sortie d'un bit représentant si l'adresse d'échec a été utilisée ou non. Cette sortie est envoyée à l'extérieur du système ainsi qu'à un multiplexeur qui permet de choisir parmi ses deux entrées, qui sont les deux adresses de sorties de la ROM. La sortie du multiplexeur correspond à l'adresse sélectionnée et représente donc la prochaine adresse à vérifier. Cette adresse est encore une fois représentée par un bus de 19 bits et est stockée dans un registre de 19 bits pour être ensuite renvoyée au module « calcul d'adresse ». Le caractère d'entrée provient soit de l'extérieur du système, soit d'un registre de huit bits stockant le caractère précédemment utilisé. Ce choix est effectué grâce à un multiplexeur dont le contrôle est effectué grâce au bit de sortie de l'encodeur, qui indique si un nœud d'échec a été utilisé ou non.

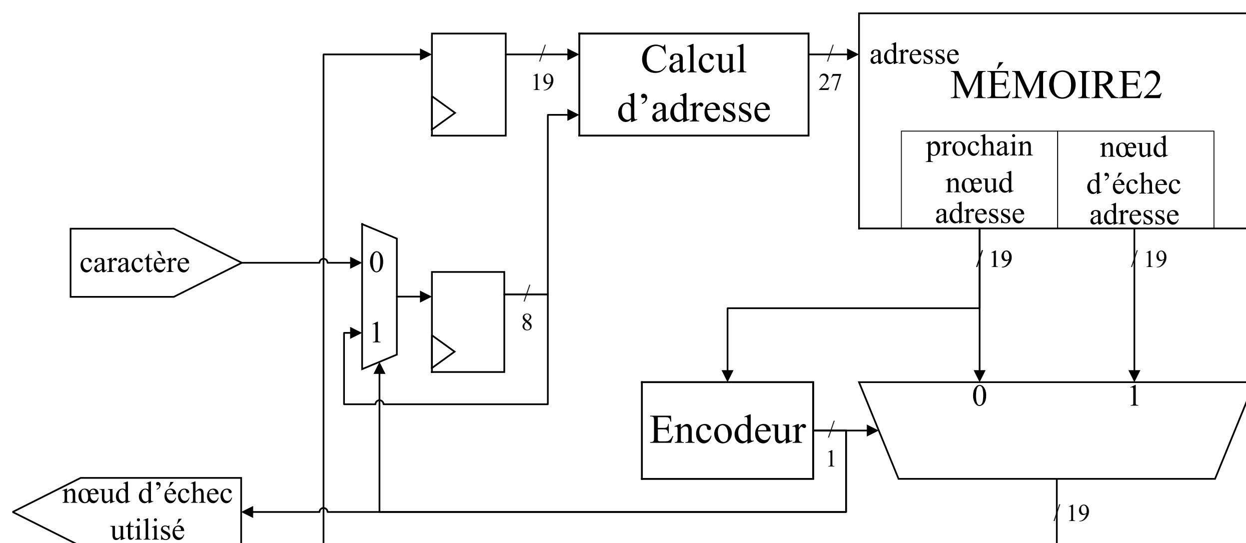


Figure 4.4: Exemple d'architecture avec des nœuds de type 2M

### 4.2.3 Système complet (types 1M et 2M)

La troisième architecture, montrée à la Figure 4.5, présente le cas où les deux types de nœuds modifiés sont utilisés. Les blocs ARCH1_0, ARCH1_1, ARCH1_n et ARCH2 représentent les architectures reliées à chaque type possible. Dans les trois premiers cas, les nœuds sont du type 1M avec 0, 1 et *n* prochains nœuds. Dans le quatrième cas, les nœuds sont du type 2M. De manière

générale, l'architecture contient $n + 1$ blocs de nœuds de type 1M et un bloc de nœud de type 2M. La variable « b » dans le diagramme suivant représente le nombre de bits requis pour représenter le type de nœud et « b » sera égal à plancher ($\log_2(n + 1) + 1$).

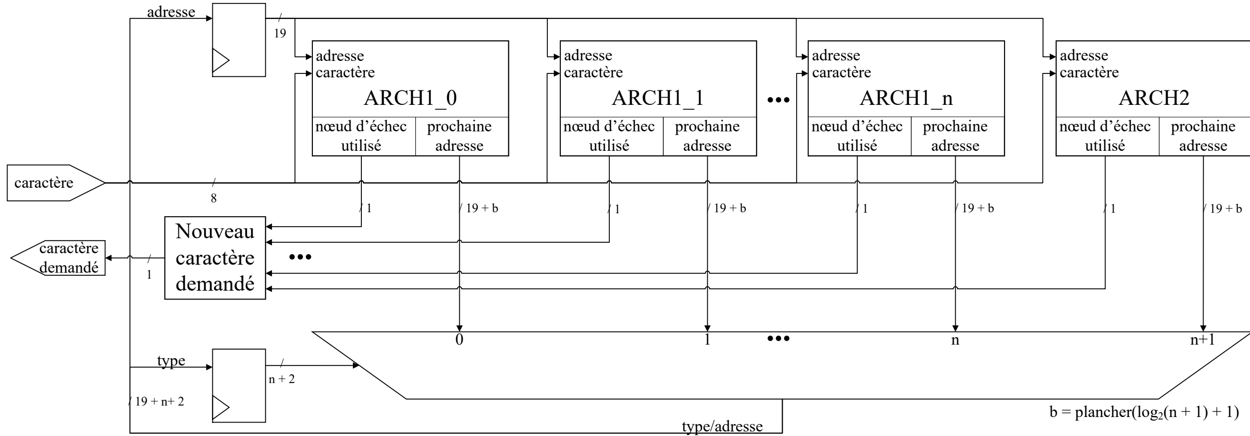


Figure 4.5: Architecture générale (types 1M et 2M)

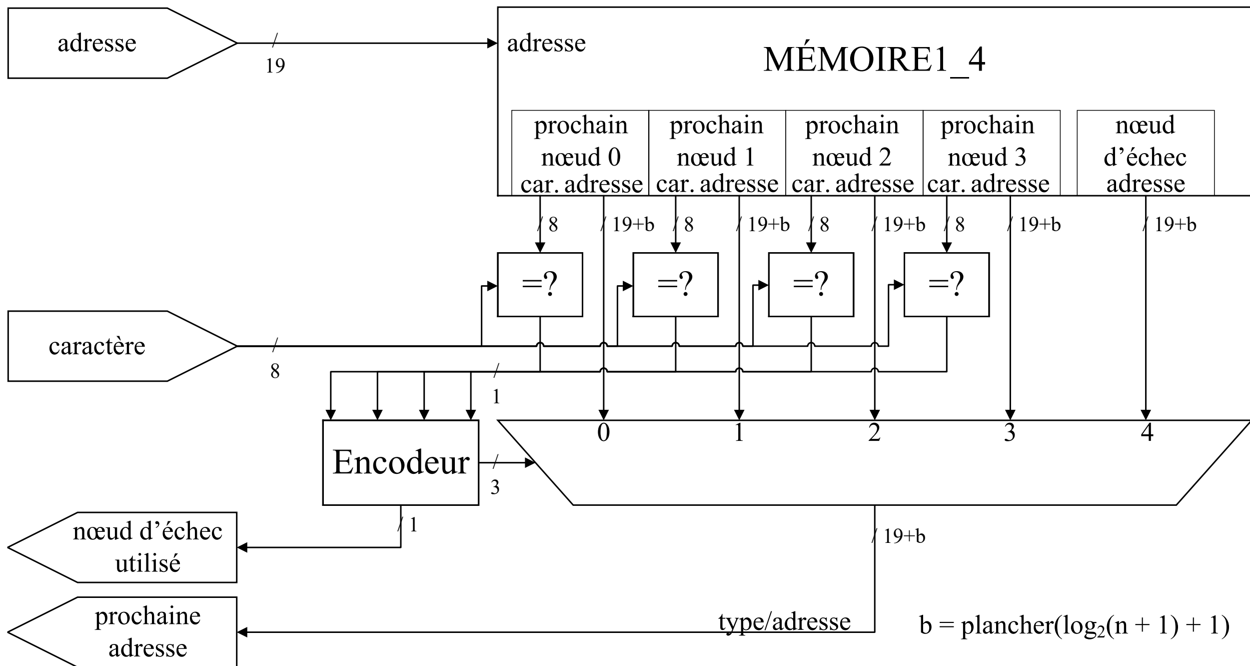


Figure 4.6: Architecture ARCH1_4

L'architecture du module ARCH1_4 est présentée à la Figure 4.6. Cette architecture contient les nœuds de type 1M avec quatre prochains nœuds. Les différences avec la Figure 4.3 sont les suivantes :

- L'adresse de 19 bits provient d'une entrée du système.
- Le caractère est valide lorsqu'il entre dans le système.
- Les adresses sortant de la mémoire comprennent aussi le type du prochain nœud.
- Dans cet exemple, les prochaines adresses sont représentées avec un bus de 19 + b bits (b bits pour le type et 19 bits pour l'adresse).
- La prochaine adresse est une sortie du système.

À la Figure 4.5, les entrées de chaque bloc ARCH sont l'adresse et le caractère d'entrée. Ils sont généralement représentés par un bus de 19 bits et par un bus de huit bits, respectivement. Le nombre de bits pour l'adresse dépend du nombre maximal d'adresses pour le type ayant le plus de nœuds. Il faut que la plus grande adresse puisse être représentée. Les sorties du bloc sont un bus de 19 + plancher ($\log_2(n + 1) + 1$) bits pour l'adresse du prochain nœud. Cette adresse contient le type du prochain nœud sur plancher ($\log_2(n + 1) + 1$) bits et l'adresse du prochain nœud sur 19 bits. La deuxième sortie est un bit représentant si l'adresse d'échec a été utilisée. Ce bit de sortie de chaque ARCH entre dans le module « prochain caractère ». Il vérifie si un des bits d'entrées représente l'utilisation de l'adresse d'échec. La sortie du module est un bit qui sort du système et qui permet de savoir si un nouveau caractère peut être envoyé au système. L'adresse de 19 + plancher ($\log_2(n + 1) + 1$) bits de chaque ARCH est envoyée à un multiplexeur. La sortie du multiplexeur est une adresse de 19 + plancher ($\log_2(n + 1) + 1$) bits. Elle est choisie grâce au signal de contrôle de trois bits qui est le type de nœud extrait de la sortie précédente. La sortie du multiplexeur sera donc séparée en deux, la première partie qui est le type de nœud est sauvegardée dans un registre de trois bits et réutilisé lors de la prochaine sélection du multiplexeur. La deuxième partie est l'adresse de 19 bits qui est sauvegardée dans un registre de 19 bits et réutilisée par tous les blocs ARCH par la suite.

# CHAPITRE 5 MÉTHODOLOGIE, RÉSULTATS ET DISCUSSIONS

Suite aux présentations et explications des deux chapitres précédents sur la version logicielle et celle matérielle, ce chapitre présente les résultats des différents scénarios de test ainsi que l'équipement utilisé. Ensuite, deux fonctions objectives (une pour chaque version) sont développées. Puis, les procédures d'exécution des tests pour les deux versions sont dévoilées. Finalement, tous les résultats ainsi que les discussions sont présentés.

## 5.1 Scénarios de test

Afin de valider les travaux de ce mémoire, deux catégories de test ont été faites, correspondant aux contributions logicielle et matérielle. Le Tableau 5.1 présente les différentes sources d'entrées qui ont été testées avec l'implémentation logicielle. On y retrouve des données du Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC) [39], des règles de Snort directement, des données générées aléatoirement et des pages internet. Pour ce qui est des tests du processeur spécialisé, un seul scénario de test a été utilisé pour cette partie, soit celui contenant de vraies données venant du MACCDC.

Tableau 5.1: Définition des scénarios de test

| **Test case** | **Chars** | **Description** |
|---|---|---|
| realData_40182 | 13616 | MACCDC |
| realData_101081 | 34048 | MACCDC |
| allIn_400 | 400 | Partie *content* des règles de Snort |
| allIn_4000 | 3994 | Partie *content* des règles de Snort |
| allIn_15000 | 14994 | Partie *content* des règles de Snort |
| allIn_100000 | 99578 | Partie *content* des règles de Snort |
| random_15000 | 14938 | Générer aléatoirement |
| cnn_html | 80550 | HTML d'une page CNN |
| wiki_canada | 108269 | Wikipédia de la page du Canada (Anglais) |
| wiki_canada_fr | 249027 | Wikipédia de la page du Canada (Français) |

## 5.2 Équipement utilisé

Lors des différents tests des versions logicielle et matérielle pour comparer les configurations des tries, une machine virtuelle avec Xubuntu 14.04, 2 processeurs et 4 Gigaoctet (Go) de RAM a été utilisée. L'ordinateur utilisé roule VMware sur Windows 8.1 avec un Intel Core i7 3.4 GHz (quatre

cœurs) et 16 Go de RAM. Lors du profilage du code C/C++, l'outil Valgrind a été utilisé pour obtenir une approximation du nombre de cycles requis pour effectuer une recherche avec l'algorithme Aho-Corasick. Lors des tests du processeur spécialisé, certains programmes supplémentaires ont aussi été utilisés. En effet, QuestaSim-64 10.3a de Mentor Graphics a servi aux diverses simulations. Pour ce qui est de la synthèse, Vivado 2015.4 de Xilinx a été brièvement choisi jusqu'à ce qu'il y ait des problèmes lors de l'initialisation des mémoires avec des fichiers textes. Par la suite, ISE Design Suite 14.4 de Xilinx fonctionnait très bien et a donc été adopté pour le reste du projet.

## 5.3 Fonctions objectives

Afin de comparer les différentes solutions et d'évaluer le compromis entre divers métriques de performance, une fonction objective pour la version logicielle et une autre pour la version matérielle doivent être établies. Les métriques de performance peuvent être la fréquence, le nombre de cycles par caractère pour la recherche, le nombre de cycles total pour la recherche, la mémoire, le nombre de défauts d'antémémoire (*cache miss*), le nombre de registres, le nombre de portes logiques, le nombre de LUT, le nombre de BRAMs, le temps de synthèse, le temps de simulation, etc.

Les fonctions objectives finalement utilisées pour les deux versions se ressemblent beaucoup. Pour ce qui est de la version logicielle, la formule (1) est utilisée où $P_1$ est la performance, C, le nombre de cycles par caractère pour faire la recherche et M, la mémoire totale utilisée en octet. Pour ce qui est de la version matérielle et de la version logicielle modifiée, la formule (2) est utilisée où $P_2$ est la performance, F la fréquence d'horloge du système en mégahertz (MHz), C, le nombre de cycles total pour faire la recherche et M, la mémoire totale utilisée en mégaoctet (Mo). Les valeurs $10^{10}$ et $10^4$ sont utilisées afin d'avoir des performances proches d'un nombre entier.

$$P_1 = \frac{10^{10}}{C \times M} \quad (1)$$

$$P_2 = \frac{10^4 \times F}{C \times M} \quad (2)$$

Les différences entre l'équation 1 et 2 sont la définition exacte de la variable C et la fréquence d'horloge du système. Cette dernière est constante dans la version logicielle. De plus, les autres

métriques disponibles ne sont pas utilisées puisqu'elles ne sont pas intéressantes, sont aussi constantes ou sont déjà inclues indirectement dans une autre variable. Par exemple, les défauts d'antémémoire sont reliés au nombre de cycles par caractère.

## 5.4 Exécution des tests

Suite au développement des différentes versions, il faut exécuter les tests appropriés du Tableau 5.1 afin de trouver quelle configuration des types de nœuds est la plus performante selon les fonctions objectives définies par (1) et (2).

### 5.4.1 Version logicielle

Lors de l'exécution des tests pour la version logicielle, plusieurs étapes ont été suivies telles que présentées à la Figure 5.1. La première étape consiste en la modification des scénarios de test (section 5.1) et des seuils *n* et *m* contrôlés par des énoncés `define` dans le code C++. Ces derniers sont choisis parmi les valeurs découvertes dans les distributions des nœuds des sections 3.1 et 3.3. Des tests exhaustifs sont donc réalisés pour tous les scénarios de test ainsi qu'avec toutes les valeurs possibles des seuils *n* et *m*. Pour ce qui est des tests avec les types 1, 2 et 3, le seuil *n* est limité à l'intervalle [-1, 8]. Par la suite, la compilation du C/C++ est effectuée sur la machine virtuelle et le programme résultant a été exécuté avec l'outil Valgrind. Cette exécution comporte la création de l'arbre ainsi que le profilage de la recherche pour le scénario de test choisi. Finalement, un fichier de résultats est créé comprenant le nombre de cycles requis pour la recherche, la mémoire totale utilisée ainsi que toutes les correspondances trouvées.

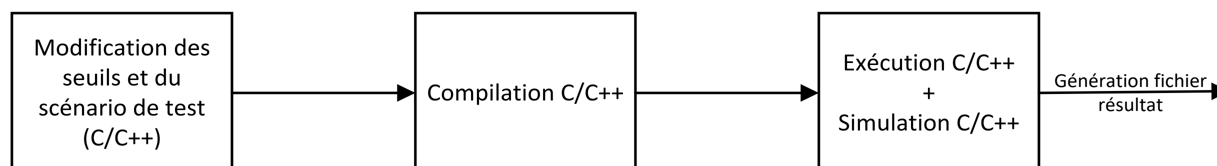


Figure 5.1: Schéma d'exécution des tests (version logicielle)

### 5.4.2 Version matérielle

Lors de l'exécution des tests pour la version matérielle, plusieurs étapes ont été suivies telles que présentées à la Figure 5.2. La première consiste en la modification du seuil *n* contrôlé par des énoncés `define` dans le code C++. Ce dernier est choisi parmi les valeurs découvertes dans la

distribution des nœuds de la section 3.1. Des tests exhaustifs sont donc réalisés pour le scénario de test choisi ainsi qu'avec toutes les valeurs possibles du seuil *n*. Par la suite, la compilation du C/C++ est effectuée et le programme résultant est exécuté avec l'outil Valgrind. Cette exécution comporte la création de l'arbre, le profilage de la recherche et la génération de plusieurs fichiers. Ces fichiers comprennent entre autres deux fichiers résultats dont le premier contient le nombre de cycles requis pour la recherche, ainsi que la mémoire totale utilisée. Le deuxième fichier contient une trace de la recherche à travers les nœuds. De plus, un fichier VHDL contenant différentes constantes, ainsi que des fichiers textes contenant tous les éléments des mémoires sont générés. Ensuite, QuestaSim 10.3a est utilisé pour compiler le code VHDL avec les données comprises dans les fichiers textes pour ainsi permettre l'exécution d'une simulation pré-synthèse. Lors de cette simulation, un deuxième fichier de résultats contenant une trace est généré. Par la suite, la synthèse du code VHDL avec les données des fichiers textes est effectuée avec l'aide de l'outil ISE 14.4. Un fichier Verilog du design est alors généré. Puis, une simulation post-synthèse conduisant à la génération d'un autre fichier résultat avec une trace est exécutée. Finalement, l'outil kdiff est utilisé afin de comparer les trois fichiers de traces pour s'assurer que les résultats sont toujours identiques.

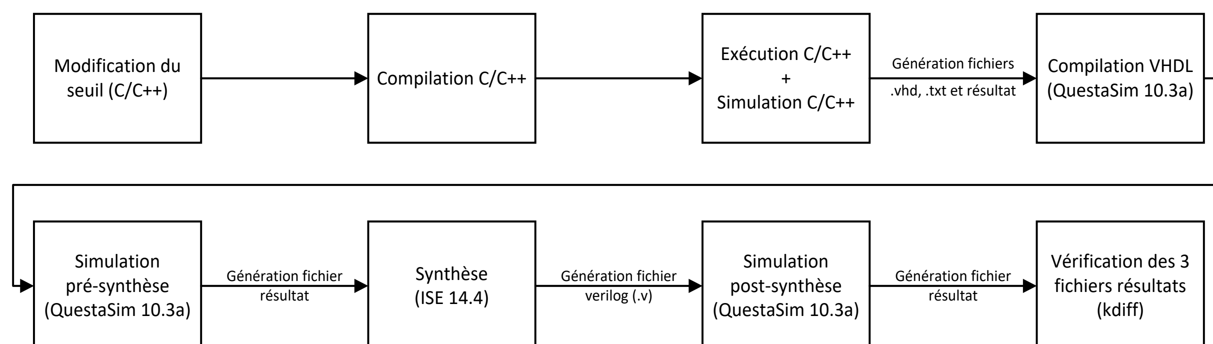


Figure 5.2: Schéma d'exécution des tests (version matérielle)

Dans les deux versions d'exécution des tests, divers fichiers Excel (.xlsx) sont générés afin de regrouper tous les résultats pertinents et d'en faire ressortir différents éléments tels que présentés à la section suivante.

## 5.5 Résultats pour la version logicielle

Les résultats présentés ci-dessous sont ceux reliés à l'expérimentation de la version logicielle. La première expérience est présentée à la Figure 5.3 et compare différentes performances calculées pour le scénario de test realData_40182 pour les valeurs de *n* dans l'intervalle [-1, 8]. Cette figure

étudie l'impact du choix du seuil $n$ sur la performance lorsque le seuil $m$ varie dans l'intervalle [-1, 254]. On constate que toutes les courbes ont la même forme et sont parallèles pour les valeurs de $m$ dans l'intervalle [40, 200] et que la valeur la plus élevée de la performance $P_1$ est lorsque $n = 1$.

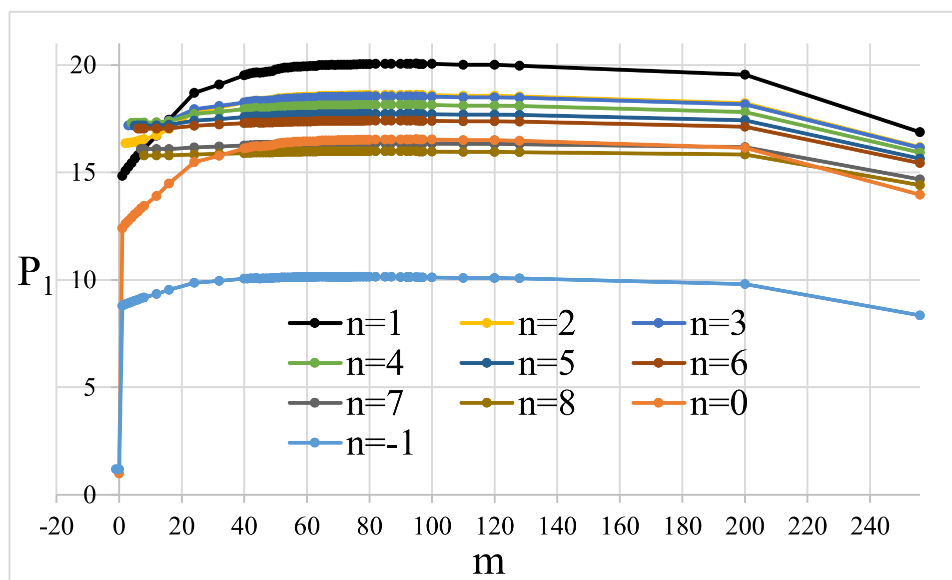


Figure 5.3: Valeur de $n$ ayant la meilleure performance avec les types de nœuds 1, 2 et 3

Pour la deuxième expérience, la performance $P_1$ est calculée en fonction de $m$ dans l'intervalle [1, 254] lorsque $n = 1$. Ceci est montré dans la Figure 5.4. Les dix scénarios de test du Tableau 5.1 y sont présents. Encore une fois, toutes les courbes ont une silhouette semblable, mais on y remarque trois regroupements de résultats. Ceux ayant les performances les plus basses proviennent de données générées aléatoirement ainsi que de pages internet. Les résultats du milieu proviennent des règles de Snort directement et les meilleures performances proviennent de vraies données du MACCDC.

La Figure 5.5 montre les mêmes données que la Figure 5.4 mais, dans ce cas, les performances sont normalisées et sont en fonction du seuil $m$. En d'autres mots, les performances de chaque scénario de test sont divisées par la meilleure performance de ce scénario de test. Encore une fois, il y a une courbe pour chaque scénario de test présent dans le Tableau 5.1. Les valeurs de $m$ varient dans l'intervalle [50, 120]. Pour toutes ces valeurs, les performances normalisées sont toutes à

moins de 1.5% du meilleur résultat. Pour la meilleure valeur de $m$, qui est approximativement autour de $m = 90$, tous les résultats sont à moins de 0.2% de la meilleure performance. Par conséquent, en utilisant les types 1, 2 et 3, la meilleure performance est atteinte lorsque $n = 1$ et $m = 90$.

À partir de ce point, seulement le scénario de test realData_40182 sera utilisé pour les tests du fait que les résultats se ressemblent beaucoup.

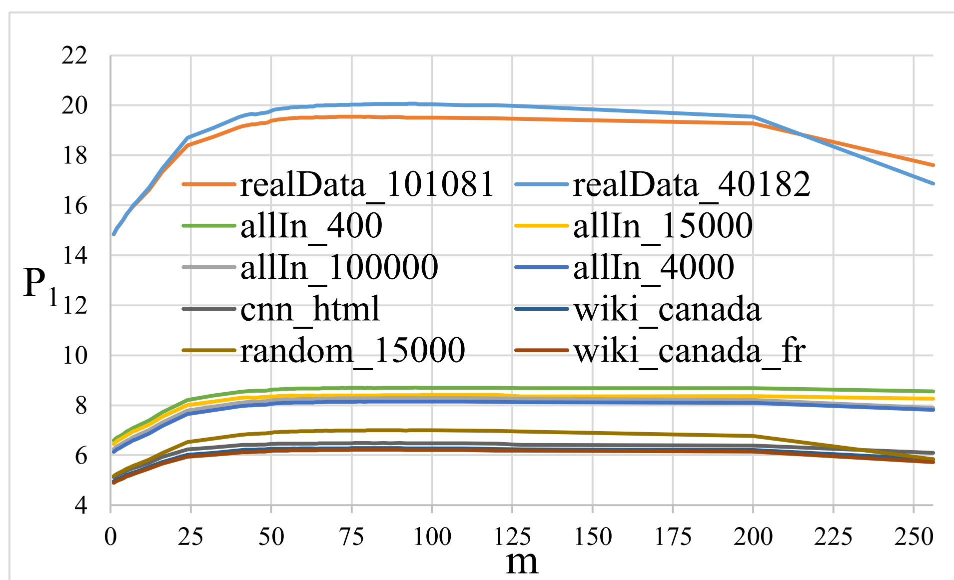


Figure 5.4: Performance lorsque $n = 1$ en utilisant les types de nœuds 1, 2 et 3

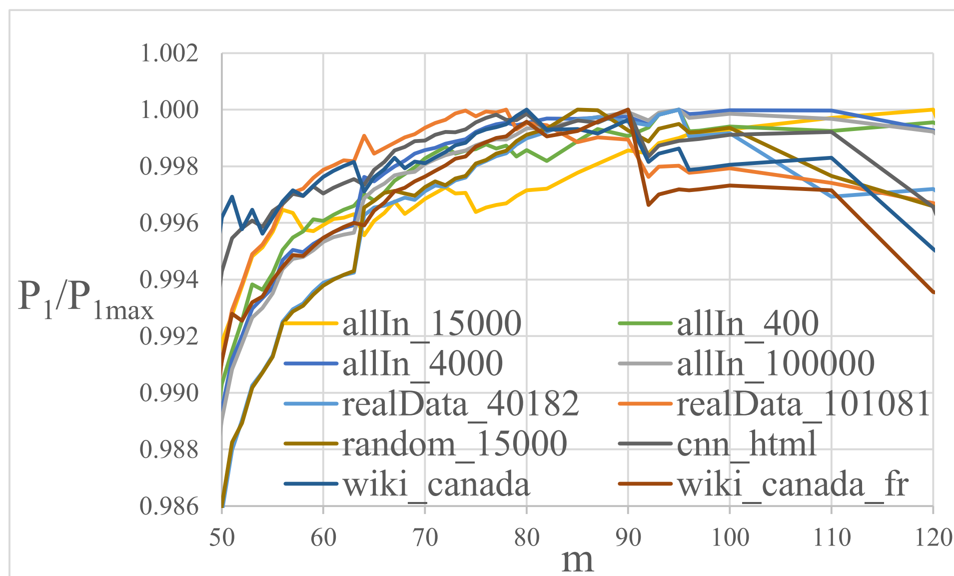


Figure 5.5: Sensibilité à la variation de $m$

La Figure 5.6 compare les performances de trois combinaisons de nœuds en fonction de $n$ dans l'intervalle [-1, 254]. Les trois combinaisons sont : les types 1 et 2; les types 1 et 3; et les types 1, 2 et 3. Les courbes de ces trois combinaisons se ressemblent beaucoup lorsque $n$ est plus grand que 0. La valeur de la performance pour les types 1 et 2 et les types 1 et 3 lorsque $n = 254$ est de 1.48. Les données pour la combinaison des types 1, 2 et 3 (en noir) sont limitées à l'intervalle [-1, 8] à cause du temps nécessaire pour la réalisation des autres expériences.

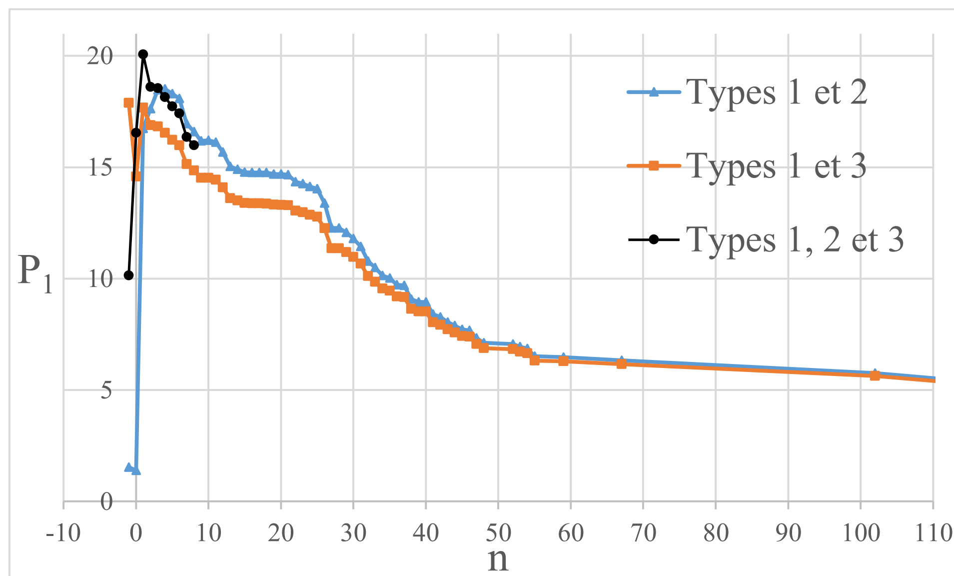


Figure 5.6: Performance

La Figure 5.7 montre un extrait du graphe de la Figure 5.6, où $n$ est dans l'intervalle [-1, 8] afin de mieux distinguer les valeurs des courbes. La première combinaison utilisant les types 1 et 3 donne son meilleur résultat lorsque $n = 1$ avec une performance $P_1$ de 17.68. La deuxième combinaison utilisant les types 1 et 2 donne son meilleur résultat lorsque $n = 4$ avec une performance $P_1$ de 18.51. La troisième combinaison utilisant les types 1, 2 et 3 donne son meilleur résultat lorsque $n = 1$ et $m = 90$ avec une performance $P_1$ de 20.06.

La Figure 5.8 présente l'utilisation mémoire totale en Mégaoctet (Mo) en fonction du seuil $n$ dans l'intervalle [-1, 16] pour les trois combinaisons vues précédemment. En utilisant les types 1 et 2, l'utilisation mémoire diminue de ~400 Mo quand $n = -1$ à ~30 Mo quand $n = 1$, ce qui représente une amélioration de plus de 10×. En utilisant les types 1 et 3, l'utilisation mémoire diminue de ~23 Mo quand $n = -1$ à ~21 Mo quand $n = 1$, ce qui ne représente pas une amélioration significative. Pour ces deux combinaisons, les valeurs de $n$ plus grandes que 16 ne sont pas présentes, mais

la mémoire totale tend à se stabiliser autour de 20 Mo. En utilisant les types 1, 2 et 3, l'utilisation mémoire diminue de ~46 Mo quand $n$ = -1 à ~21 Mo quand $n$ = 1, ce qui représente une amélioration de plus de 2×.

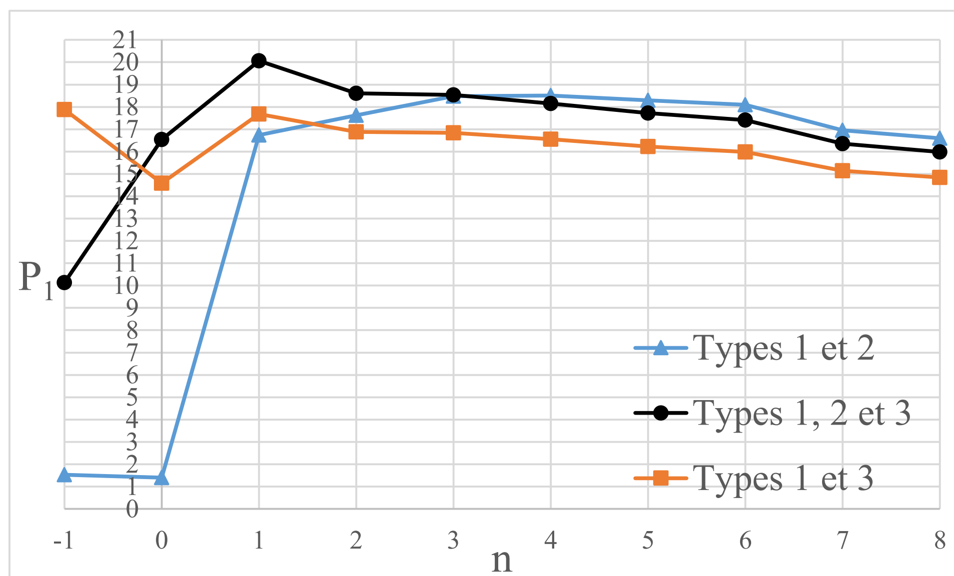


Figure 5.7: Performance avec agrandissement

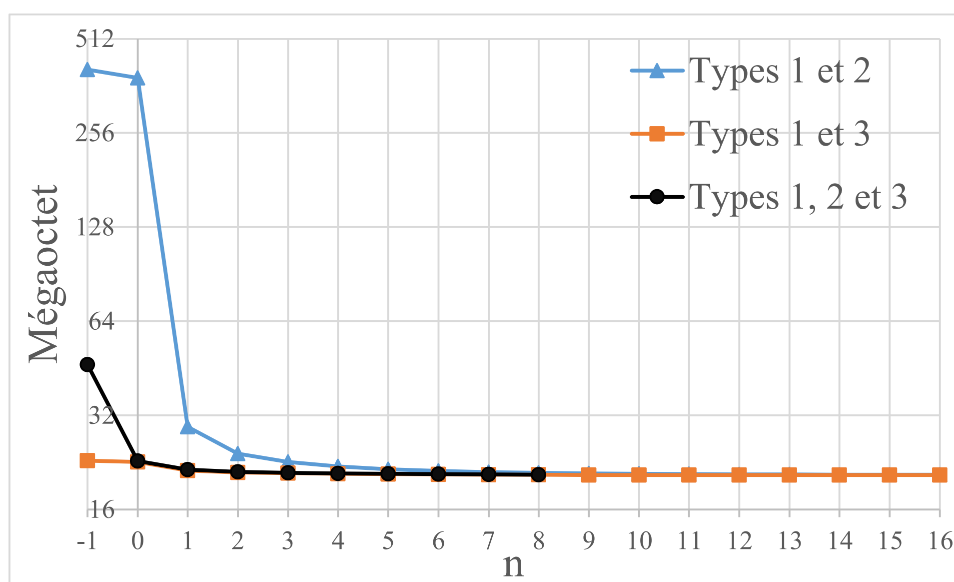


Figure 5.8: Utilisation mémoire

La Figure 5.9 montre le nombre de cycles par caractère requis pour la recherche en fonction du seuil $n$ dans l'intervalle [-1, 70]. Le graphique ne montre pas certaines valeurs : lorsque $n$ = 102, la

combinaison des types 1 et 2 requiert 84 cycles et la combinaison des types 1 et 3 requiert 87 cycles; lorsque $n$ = 254, ces deux combinaisons requièrent 329 cycles. Encore une fois, les courbes sont semblables.

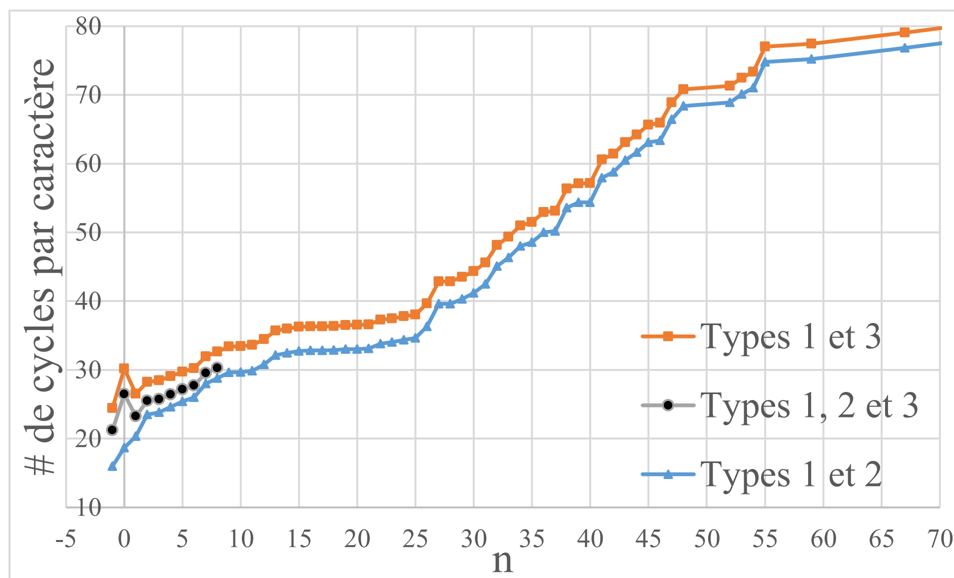


Figure 5.9: Nombre de cycles par caractère

## 5.6 Résultats pour la version matérielle

Les résultats présentés ci-dessous sont ceux reliés à l'expérimentation de la version matérielle. La Figure 5.10 présente le temps de synthèse en seconde en fonction du seuil $n$ dans l'intervalle [1, 102] en utilisant le programme ISE Design Suite 14.4 de Xilinx. Le pire cas, lorsque $n$ = 1, a un temps de synthèse de 220458 secondes, soit plus de 61 heures. Ce pire cas s'explique du fait qu'il y a beaucoup de nœuds du type 2M et que la synthèse de ce type requiert plus de temps et de ressources lorsqu'en grande quantité. À partir de $n$ = 4, le temps de synthèse se stabilise autour de 15300 secondes, soit 4 heures et 15 minutes.

La Figure 5.11 présente le nombre de BRAM utilisé après synthèse en fonction du seuil $n$ dans l'intervalle [1, 102] pour le FPGA XC7V2000T de Xilinx. La ligne droite (orange) représente le nombre maximal de BRAM disponible dans ce FPGA, soit 1292. Dans le pire cas, lorsque $n$ = 1, le nombre total de BRAM utilisé est de 2112, ce qui représente 163% des BRAM disponibles. Le meilleur cas utilise 811 BRAM, soit 62%.

La Figure 5.12 présente la fréquence de l'horloge du système après la synthèse en fonction du seuil *n* dans l'intervalle [1, 102]. Le meilleur résultat est lorsque *n* = 1 parce que la fréquence est reliée aux nombres d'éléments requis pour le design. En général, lorsque la surface utilisée augmente, la fréquence diminue. Lorsque *n* = 1, le design contient seulement trois types de nœuds (1_0, 1_1 et 2) et ne contient donc qu'un seul comparateur, les multiplexeurs sont plus simples, etc. En d'autres mots, plus le système est simple, plus la fréquence est élevée.

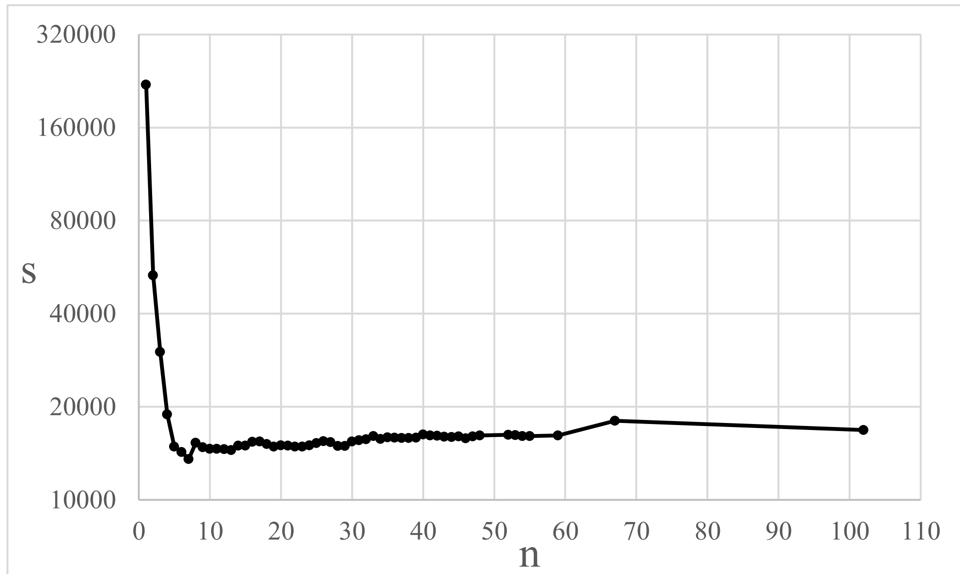


Figure 5.10: Temps de synthèse

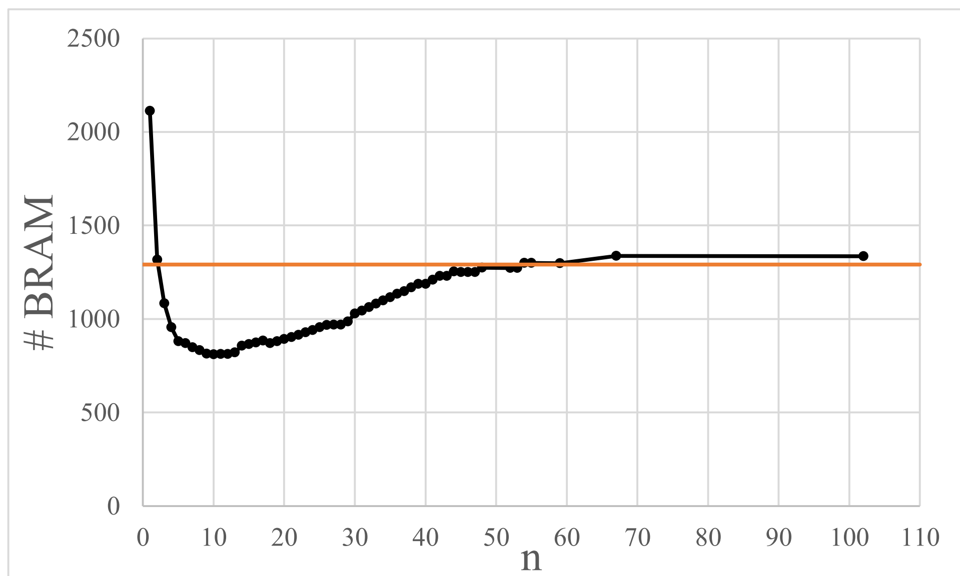


Figure 5.11: Nombre de BRAM utilisé avec Xilinx XC7V2000T

La Figure 5.13 présente le nombre de registres et le nombre de LUT nécessaires en fonction du seuil $n$ dans l'intervalle [1, 102]. Les deux courbes sont semblables. En effet, les courbes sont linéaires jusqu'à $n = 67$, mais les valeurs ne sont plus conformes lorsque $n = 102$. Ce changement a lieu parce qu'il n'y a qu'un seul nœud avec 102 prochains nœuds et que la modification de ce nœud n'est pas significative. Les designs utilisés dans ce mémoire ne requièrent pas beaucoup de registres ni de LUT. En effet, le FPGA XC7V2000T de Xilinx utilisé a plus de deux millions de registres et au maximum seulement 500 sont utilisés (<0.1%). Il a aussi plus d'un million de LUT et au maximum seulement 86000 sont utilisés (~7%).

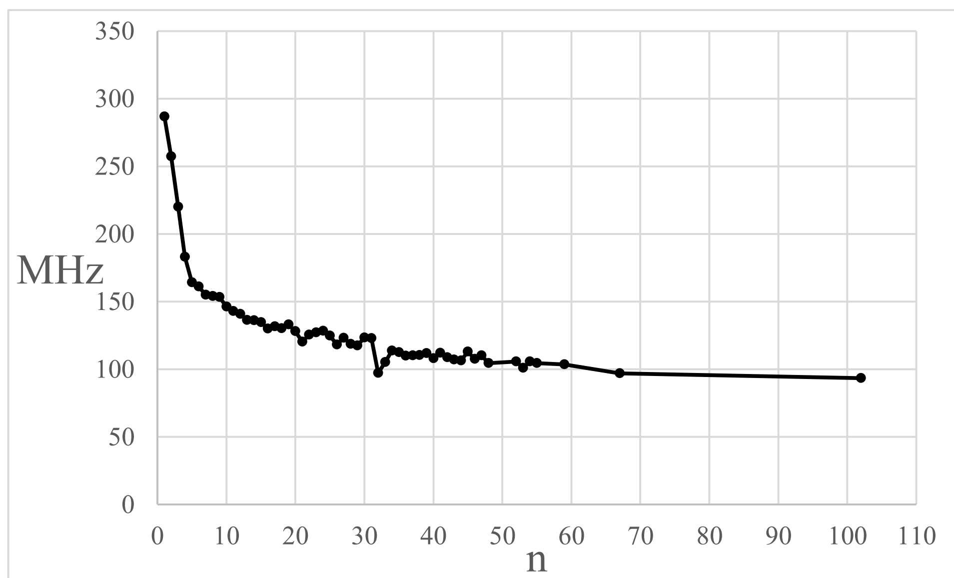


Figure 5.12: Fréquence de l'horloge

La Figure 5.14 présente la comparaison de mémoire totale utilisée en fonction du seuil $n$ dans l'intervalle [1, 102] pour la version logicielle modifiée et la version matérielle. Les courbes sont similaires. La version logicielle modifiée utilise 19.5 Mo lorsque $n = 1$, et tend vers 10.6 Mo par la suite, ce qui représente une amélioration d'environ 46% (presque 2× mieux). Pour ce qui est de la version matérielle, elle utilise 8.5 Mo lorsque $n = 1$, et tend vers 3 Mo par la suite, ce qui représente une amélioration d'environ 65% (presque 3× mieux). En général, le fait d'utiliser la version matérielle requiert environ 3× moins de mémoire par rapport à la version logicielle modifiée.

La Figure 5.15 présente la comparaison du temps de recherche en fonction du seuil $n$ dans l'intervalle [1, 102] pour la version logicielle modifiée et la version matérielle. Dans l'intervalle [1, 3], la version matérielle est légèrement plus rapide. Les courbes sont similaires dans l'intervalle

[4, 27]. Par la suite, les deux courbes augmentent plutôt linéairement, mais celle de la version logicielle modifiée a une pente environ 7× plus grande.

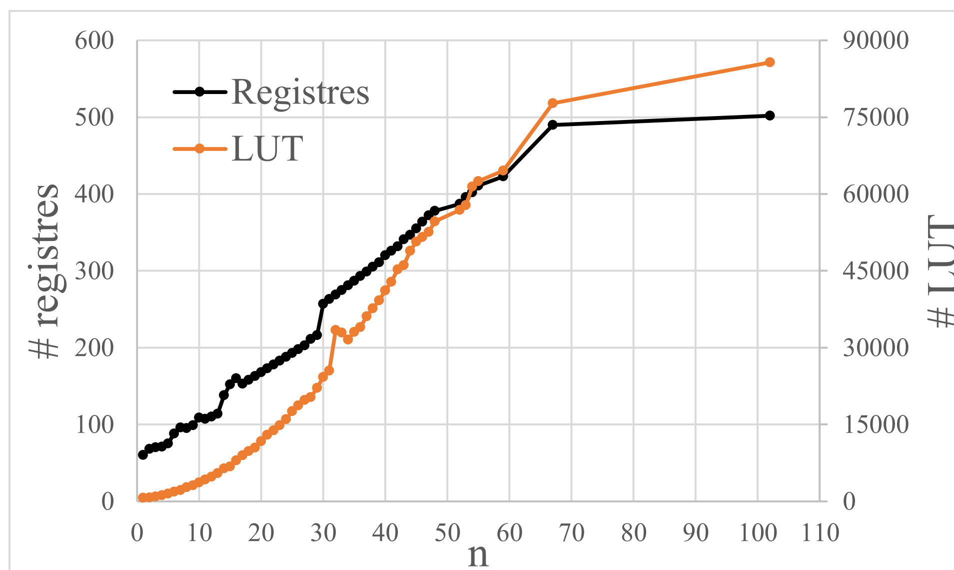


Figure 5.13: Nombre de registres et nombre de LUT

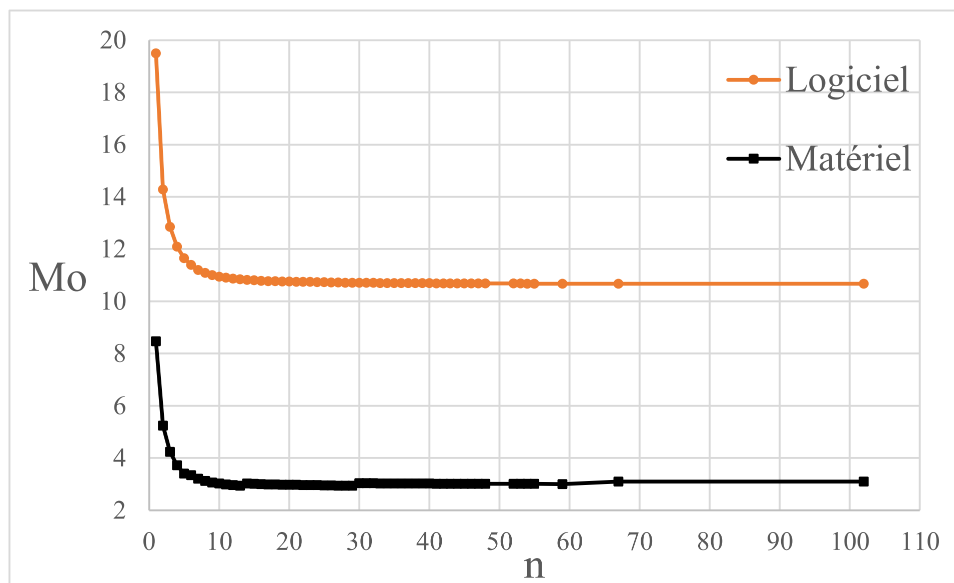


Figure 5.14: Utilisation mémoire

La Figure 5.16 compare la performance $P_2$ en fonction du seuil $n$ dans l'intervalle [1, 102] pour la version matérielle et la version logicielle modifiée. Dans les deux cas, la pire performance apparaît lorsque $n = 102$ et la version matérielle est plus performante d'environ 6×. Pour ce qui est des

meilleures performances, avec la version matérielle, elle apparaît lorsque $n = 3$ avec une performance $P_2$ de 9.59. Avec la version logicielle modifiée, la meilleure performance apparaît lorsque $n = 6$ avec $P_2$ égale à 2.71. Le fait d'utiliser la version matérielle apporte une amélioration de la performance de 3.5× par rapport à la version logicielle modifiée. Dans les deux courbes, depuis les points maximaux, les performances diminuent plutôt linéairement.

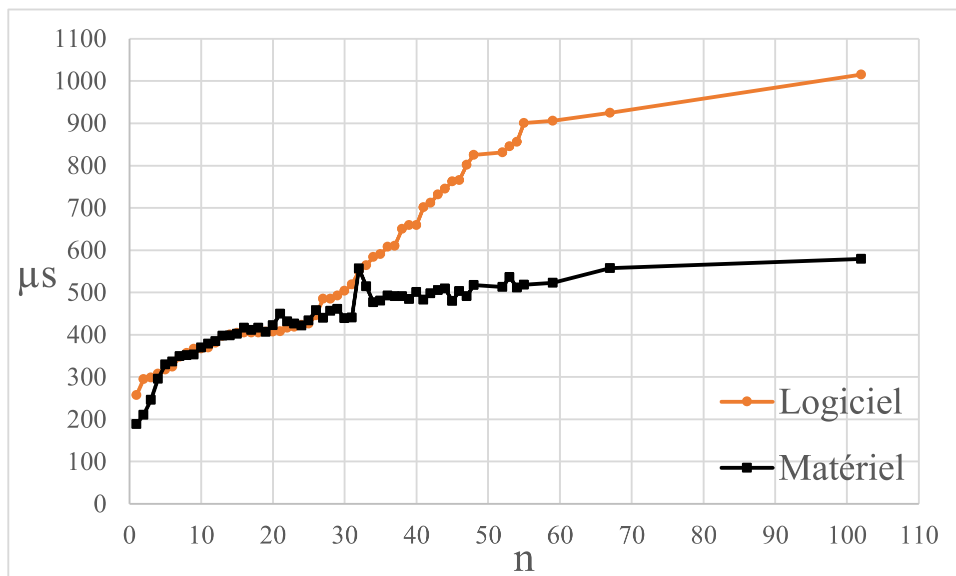


Figure 5.15: Temps de recherche

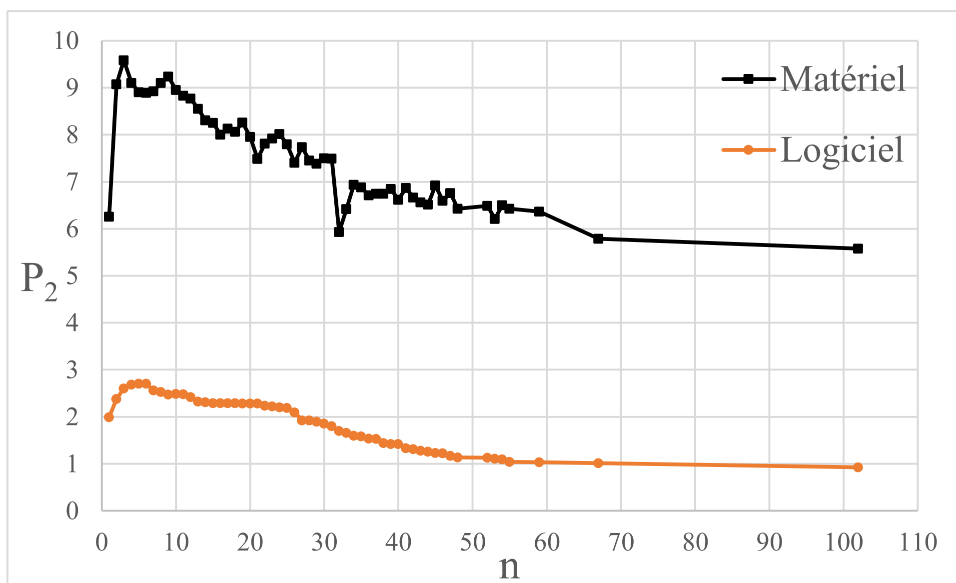


Figure 5.16: Performance

## 5.7 Discussion

Dans les sous-sections précédentes, les résultats de la version logicielle, de la version matérielle et de la version logicielle modifiée ont été présentés. Cette sous-section effectue une comparaison entre les résultats obtenus dans ce mémoire et ceux présents dans la littérature. Le Tableau 5.2 présente cette comparaison.

Notre comparaison est effectuée à partir des critères suivants :

- la source de la publication;
- la version de Snort utilisée;
- une description du test;
- le nombre de règles de Snort utilisées;
- le nombre de chaînes de caractères utilisées qui sont présentes dans les règles;
- le nombre total de caractères présents dans les chaînes de caractères;
- le nombre de nœuds présents après la création de la FSM;
- la mémoire totale (Mo) utilisée pour stocker la FSM;
- la mémoire requise par nœud (octets);
- la mémoire requise par caractère (octets);
- le débit (Gigabit par seconde - Gbps) si tu FPGA est utilisé; et,
- le débit (Gbps) des autres cas.

La majorité des auteurs ne fournissent pas toutes ces informations, et, parmi celles présentes, les valeurs ne coïncident pas toujours. Par exemple, les versions de Snort utilisées ne sont pas toujours énoncées; et, pour deux mêmes versions de Snort, le nombre de chaînes de caractères varie beaucoup. Ces différences pourraient venir du fait que certains travaux utilisent des sous-ensembles des règles de Snort. Les différences reliées au nombre de nœuds dépendent du nombre de caractères à représenter, mais aussi de la représentation des nœuds choisie par les auteurs. Par exemple, si un auteur choisit de compresser les chemins, il y aura moins de nœuds. De plus, les auteurs n'ont probablement pas toujours la même définition de ce qu'est un nœud et des données qu'il contient.

De plus, certains auteurs fournissent seulement la mémoire totale utilisée et d'autres seulement le débit. Dû au fait qu'il manque des informations, les comparaisons entre les différents auteurs sont limitées.

Les résultats de la consommation de mémoire requise ainsi que du débit sont présentés et comparés. Pour la consommation mémoire, les deux colonnes les plus intéressantes sont celles de la mémoire requise par nœud et de la mémoire requise par caractère. Dans les deux cas, des implémentations logicielles et matérielles sont comparées. Le premier cas présenté est celui de la mémoire requise par nœud. Lors des expériences purement logicielles, la plus grande consommation mémoire est de 1070 octets par nœud. Lorsque la performance est la plus élevée ($n = 1$ et $m = 90$), la consommation diminue à 56 octets par nœud. Lors des expériences avec la version logicielle modifiée et la version matérielle, lorsque la meilleure performance de chaque version est atteinte, la consommation mémoire est de 30 octets par nœud et de 11 octets par nœud respectivement. La plus faible consommation mémoire par nœud étant celle de la version matérielle.

Le deuxième cas présenté est celui de la mémoire requise par caractère. Lors des expériences purement logicielles de ce mémoire, la plus grande consommation mémoire est de 789.6 octets par caractère. Lorsque la performance est la plus élevée ($n = 1$ et $m = 90$), la consommation diminue à 41.5 octets par caractère. Lors des expériences avec la version logicielle modifiée et la version matérielle, lorsque la meilleure performance de chaque version est atteinte, la consommation mémoire est de 22 octets par caractère et de 8.2 octets par caractère, respectivement. La plus faible consommation mémoire par caractère est celle de Pao et al. [37] avec 2.7 octets par caractère en utilisant un algorithme Aho-Corasick pipeliné. Une autre consommation de mémoire semblable est celle de Bremler-Barr et al. [20] avec 3.2 octets par caractère en utilisant l'équivalent du type 1 avec une compression des chemins.

Pour ce qui est du débit du système, les résultats de ce travail sont inférieurs à ceux de la littérature. En effet, lors des expériences avec la version logicielle modifiée, lorsque la meilleure performance est atteinte, le débit est de 0.3 Gbps. Lors des expériences avec la version matérielle sur FPGA, lorsque la meilleure performance de la simulation est atteinte, le débit est de 0.4 Gbps. Les débits les plus élevés de la littérature sont: le NFA de Chen et al. [34] sur FPGA avec 16.8 Gbps, et l'utilisation de GPU et des STT de Tran et al. [24] avec 15 Gbps. Par contre, ces deux auteurs ne montrent pas leur consommation mémoire. Les plus hauts débits suivants sont : le bit-split de Tan

et al. [26] sur FPGA avec 10 Gbps et le bitmap de Tuck et al. [18] sur ASIC avec 7.8 Gbps. Pour ces deux auteurs, leur consommation mémoire est supérieure à celle de ce travail, soit 4× plus et 7× plus, respectivement. Par contre, leur ratio débit/mémoire est plus élevé.

Lors des tests de la version matérielle, en utilisant les nœuds de type 1M et 2M, la consommation mémoire diminue de 8.5 Mo à 3 Mo, ce qui représente une amélioration de presque 3×. La consommation mémoire se situe très bien par rapport à la littérature. Lors des tests de la version logicielle modifiée, en utilisant les nœuds de type 1 et 2, la consommation mémoire diminue de 19.5 Mo à 10.6 Mo, ce qui représente une amélioration de presque 2×. Lors de la meilleure performance de la version logicielle modifiée, la consommation mémoire est de 11 Mo, ce qui se traduit par 30 octets par nœud et lors de la meilleure performance de la version matérielle, la consommation mémoire est de 4 Mo, ce qui se traduit par 11 octets par nœud. De manière générale, le fait d'utiliser la version matérielle requiert environ 3× moins de mémoire par rapport à la version logicielle modifiée. Pour ce qui est du débit, il augmente de 0.3 Gbps pour la version logicielle modifiée à 0.4 Gbps pour la version matérielle.

Tableau 5.2: Comparaison avec la littérature

| Publication | Version de Snort | Tests | Règles | Chaînes de caractères | Caractères | Nœuds | Mémoire (Mo) | Mem/nœud (octets) | Mem/caractère (octets) | Débit FPGA (Gbps) | Débit autre (Gbps) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tuck, 2004 [18] | Nov. 2000 | Aho-Corasick non-optimisée | | | | | 40.20 | | | | |
| | | Bitmap | | | | | 2.00 | | | | |
| | | Bitmap + compression des chemins | | | | | 0.70 | | | | |
| | Juin 2003 | Aho-Corasick non-optimisée | | 1533 | 19124 | | 53.10 | | 2776.6 | 3.0 | 6.0 |
| | | Bitmap | | 1533 | 19124 | | 2.80 | | 146.4 | 6.5 | 7.8 |
| | | Bitmap + compression des chemins | | 1533 | 19124 | | 1.10 | | 57.5 | 6.5 | 7.8 |
| Tan, 2005 [26] | | Bit-split | | 1000 | | 10000 | 0.40 | 40 | | 10.0 | |
| Piyachon, 2006 [30] | Déc. 2005 (2.4.x?) | Traditionnel | 4219 | | 42028 | | 84.15 | | 2002.2 | | |
| | | Bit-byte level | 4219 | | 42028 | | 0.27 | | 6.5 | | 5.0 |
| Dimopoulos, 2007 [22] | (2.4) | Split-AC | 3000 | | 24033 | | 0.19 | | 8.1 | 0.9 | |
| | | Bit-split de Tan, 2005 | | | 12812 | | 0.40 | | 32.6 | | |
| Piyachon , 2008 [31] | Sept. 2007 (2.8.x?) | LTT + CLT | | 5892 | 91575 | | 1.38 | | | | |
| | | Version de base | | 5892 | 91575 | | 150.00 | | 1638.0 | | |
| | Avr. 2004 | LTT + CLT | | | | | 0.32 | | | | |
| Smith, 2008 [32] | Mars 2007 (2.7.x?) | XFA | 1450 | | | 41994 | 46.50 | 1107 | | | |
| Zha, 2008 [19] | | Bitmap + compression des chemins | | 2430 | | | 0.43 | | | | |
| Pao, 2008 [37] | | Version de bit-split de Tan, 2005 | | | 12812 | | 0.30 | | 23.3 | | |
| | | P-AC | | 4434 | 84015 | | 0.23 | | 2.7 | 8.8 | |
| Lin, 2010 [28] | (2.4) | GPU avec plusieurs petites FSM | | 994 | 22776 | 8285 | 0.11 | 14 | 5.0 | | 3.9 |
| Tumeo, 2010 [36] | | GPU + 1 paquet par fil d'exécution | | | | | | | | | 6.7 |
| Shenoy, 2011 [23] | Sept. 2007 (2.8.x?) | Version de base | 3816 | 23653 | | 42000 | 44.00 | 1048 | | | |
| | | Avec leur deux types de stockage | 3816 | 23653 | | 42000 | 5.10 | 121 | | | 3.4 |
| | Avr. 2010 (2.8.x?) | Avec leur deux types de stockage | | 40678 | | | | | | | |
| Bremler-Barr, 2011 [20] | 2010 (2.x?) | Version de base | | 31094 | 105246 | 77182 | 75.15 | 974 | 714.0 | | 0.8 |
| | | Tableau de pairs symbole-état + compression des chemins | | 31094 | 105263 | 11927 | 0.34 | 29 | 3.2 | | 0.1 |
| Chen, 2011 [33] | Juill. 2010 (2.8.6.1) | Arbre suffixe | | 1485 | | | 40.00 | | | | |
| Tran, 2012 [24] | | GPU + STT | | | | | | | | | 15.0 |
| Chen, 2013 [34] | | NFA | | 1000 | 13566 | | | | | 16.8 | |
| Ce travail | Oct. 2014 (2.9.7) | Logiciel, pire utilisation mémoire | 31133 | 26328 | 516627 | 381302 | 407.93 | 1070 | 789.6 | | |
| | | Logiciel, meilleure performance | 31133 | 26328 | 516627 | 381302 | 21.43 | 56 | 41.5 | | |
| | | Logiciel simplifié, meilleure performance | 31133 | 26328 | 516627 | 381302 | 11.39 | 30 | 22.0 | | 0.3 |
| | | Matériel, meilleure performance | 31133 | 26328 | 516627 | 381302 | 4.24 | 11 | 8.2 | 0.4 | |

# CHAPITRE 6 CONCLUSION ET TRAVAUX FUTURS

## 6.1 Synthèse des travaux et contributions

L’objectif général de ce travail était de proposer des méthodes et architectures efficaces pour l’implémentation d’un IDS utilisant l’algorithme Aho-Corasick.

Tout d’abord, une implémentation logicielle pour des processeurs à usage général pour la détection d’intrusions dans des CTD a été développée. Trois objectifs ont été complétés. Premièrement, il fallait identifier une application de détection d’intrusions. Snort a été sélectionné puisque c'est un IDS communément utilisé et l'algorithme Aho-Corasick est utilisé dans la version 2.9.7 [6].

Deuxièmement, il fallait proposer des versions plus efficaces de l’algorithme qui tiennent compte des ressources mémoire, des différentes représentations des informations en mémoire et de l’exécution séquentielle de ces processeurs. Pour cela, trois types de structures de données de nœuds ont été proposés. Le nœud de type 1 est composé d’un tableau de structures contenant le nombre spécifique de caractères et leur pointeur vers le prochain nœud correspondant. La recherche se fait en temps linéaire dans le pire cas. Le nœud de type 2 est composé d’un tableau contenant 256 pointeurs vers les prochains nœuds et le caractère d’entrée est utilisé comme index. Ce type de nœud mène à du gaspillage de mémoire. La recherche se fait en temps constant. Le nœud de type 3 est composé d’un tableau contenant des pointeurs, d’un caractère ayant la plus petite valeur ASCII présente dans le tableau et d’un nombre représentant la taille du tableau. La consommation mémoire est diminuée par rapport au nœud de type 2. La recherche se fait encore en temps constant.

Troisièmement, il fallait vérifier le fonctionnement correct de l’application grâce à des scénarios de test utilisant des données variées, mesurer leur consommation de ressources et extraire leurs fonctions objectives de performances. Pour cela, plusieurs scénarios de tests ont été utilisés incluant de vraies données réseau. La fonction objective de performance proposée fait un compromis entre le nombre de cycles pour effectuer la recherche et la mémoire consommée. En utilisant les nœuds de type 1, 2 et 3, lorsque $n = 1$ et $m = 90$, la consommation mémoire diminue de 407 Mo à 21 Mo pour une FSM d’environ $381 \times 10^3$ nœuds, ce qui représente une amélioration d’environ 20×. La consommation mémoire par nœud a donc diminué de 1070 octets à 56 octets par nœud et se situe bien par rapport à la littérature.

Ensuite, un processeur particularisé pour la détection d'intrusions dans des CTD a été développé. Trois objectifs ont été complétés. Premièrement, pour la même application que précédemment, il fallait proposer des versions plus efficaces, qui tiennent compte des ressources de calcul disponibles, des différentes représentations des informations en mémoire et du parallélisme inhérent des FPGA. Pour cela, deux types de structure de données de nœuds modifiés ont été proposés. Le nœud de type 1M est composé de trois tableaux contenant le nombre spécifique de caractères, leur type et leur pointeur vers le prochain nœud correspondant. Lors de la recherche, il faut passer à travers tous les éléments des trois tableaux en parallèle et la comparaison des caractères se fait avec des comparateurs en parallèle. La recherche se fait donc en temps constant. Le nœud de type 2M est composé de deux tableaux contenant 256 types de nœuds et pointeurs vers les prochains nœuds. Le caractère d'entrée est utilisé comme index. Ce type de nœud mène à du gaspillage de mémoire. La recherche se fait en temps constant.

Deuxièmement, il fallait développer, pour l'application proposée, des chemins de données et des unités de contrôle pour des processeurs particularisés dans un langage de description matérielle comme VHDL et les implémenter sur FPGA. En effet, trois architectures matérielles ont été proposées. La première utilise uniquement les nœuds de type 1M, la deuxième utilise uniquement les nœuds de type 2M et finalement, la troisième utilise les nœuds de type 1M et 2M. De plus, une implémentation logicielle modifiée a été développée afin de pouvoir comparer les résultats.

Troisièmement, il fallait vérifier le fonctionnement correct de l'application grâce à des scénarios de test utilisant des données variées, mesurer leur débit de traitement et consommation de ressources et extraire leurs fonctions objectives de performances. Pour cela, un scénario de test a été utilisé incluant de vraies données réseau. La fonction objective de performance proposée fait un compromis entre la fréquence du système, le nombre de cycles pour effectuer la recherche et la mémoire consommée. Pour ce qui est de la version matérielle, en utilisant les nœuds de type 1M et 2M, la consommation mémoire diminue de 8.5 Mo à 3 Mo pour une FSM d'environ $381\times10^3$ nœuds, ce qui représente une amélioration de presque 3×. La consommation mémoire se situe très bien par rapport à la littérature. Pour ce qui est de la version logicielle modifiée, en utilisant les nœuds de type 1 et 2, la consommation mémoire diminue de 19.5 Mo à 10.6 Mo pour un même FSM, ce qui représente une amélioration de presque 2×. Lors de la meilleure performance de la version logicielle modifiée, la consommation mémoire est de 11 Mo, ce qui se traduit par 30 octets par nœud et lors de la meilleure performance de la version matérielle, la consommation mémoire

est de 4 Mo, ce qui se traduit par 11 octets par nœud. De manière générale, le fait d'utiliser la version matérielle requiert environ 3× moins de mémoire par rapport à la version logicielle modifiée. Pour ce qui est du débit, il augmente de 0.3 Gbps pour la version logicielle modifiée à 0.4 Gbps pour la version matérielle.

Finalement, une dernière contribution de ce mémoire est présentée. Cette contribution est un article publié au ACM/IEEE Symposium on Architectures for Networking and Communications Systems 2016 (ANCS'16) [14]. Cet article montre qu'il est possible de choisir un paramètre afin d'obtenir un produit mémoire × cycles optimal. Ce produit mène à une amélioration de performance de 12×. Ils proposent une configuration de nœuds parmi les nœuds de type 1 et 2.

## 6.2 Limitations

Suite au développement des diverses versions, plusieurs limitations ont été observées. Tout d'abord, on constate une limitation de la solution logicielle. En effet, lors de l'implémentation de l'algorithme Aho-Corasick, uniquement les caractères ASCII étendue ou UTF-8 ont été utilisés. Par contre, si les caractères d'autres langues devaient être employés, il faudrait utiliser l'encodage UTF-16 et ses 65536 caractères. Ce changement requerrait diverses modifications pour les types 1 et 2. Cette limitation est aussi applicable à la solution matérielle où les architectures devraient aussi être modifiées.

Ensuite, deux limitations de la solution matérielle sont observées. Premièrement, la reconfiguration du FPGA serait problématique si le nombre de règles augmentait. En effet, puisque des ROM sont utilisées dans toutes les mémoires de l'architecture matérielle, si une règle était ajoutée, tout le processus de synthèse et d'implémentation devrait être refait. Tel que vu précédemment, ce processus a une durée de plusieurs heures. Deuxièmement, un débalancement entre la mémoire BRAM et la logique est présent. Effectivement, la performance optimale de la solution matérielle utilise moins de 0.1% de la logique présente dans le FPGA, mais environ 80% des BRAM est utilisé. Sachant qu'un des plus gros FPGA de Xilinx a été employé, si le nombre de règles augmentait, tous les nœuds du nouvel arbre généré ne rentreraient pas dans la mémoire.

Finalement, deux limitations dans les expériences sont constatées. Premièrement, pour la version logicielle, tous les différents tests auraient dû être exécutés. En effet, tous les tests n'ont pas été exécutés pour les types 1, 2 et 3 en raison d'une contrainte de temps. Il faudrait prévoir au moins

50 jours. Ces derniers pourraient donner des résultats surprenants, bien que nous ne le croyons pas. Deuxièmement, pour la version matérielle, uniquement un seul scénario de test a été choisi. Effectivement, tous les scénarios présents dans la version logicielle devraient être exécutés afin d'obtenir des résultats plus complets et si possible, indépendants des entrées.

## 6.3 Travaux futurs

Suite aux travaux présentés dans ce mémoire, plusieurs modifications pourraient être apportées. Une modification possible serait pour le nœud de type 2, lorsque le nœud d'échec est emprunté pour retourner au nœud racine. En effet, il serait intéressant de directement calculer la prochaine adresse lors de la création de l'arbre pour ne pas devoir passer par le nœud racine et de l'entreposer dans le tableau directement. En d'autres mots, on pourrait calculer la fonction d'échec et l'incorporer directement dans le tableau de 256 éléments. Ceci permettrait d'aller directement au nœud voulu selon le caractère d'entrée au lieu d'aller au nœud racine et de revérifier le caractère courant. Il n'y aurait donc plus de pointeurs NULL dans les tableaux de prochains nœuds. Cette modification permettrait aussi d'enlever le nœud d'échec de tous les nœuds de types 2 et pourrait possiblement sauver un cycle de recherche à chaque fois que la recherche passe par le nœud racine alors que le caractère recherché se trouve au deuxième niveau de l'automate. Par exemple, dans la Figure 6.1, l'automate « a » représente ce qui est fait actuellement avec un nœud d'échec et un tableau de 256 éléments pour chaque nœud. L'automate « b » représente la modification où les nœuds d'échecs seraient incorporés dans le tableau de 256 éléments (les courbes orange).

Différentes modifications possibles pour la version matérielle serait d'incorporer le nœud de type 3 présenté à la section 3.2.3, de faire des tests avec plusieurs scénarios de test tel que présenté à la section 5.1, de représenter la mémoire différemment comme, par exemple, en concaténant différents éléments des nœuds, etc.

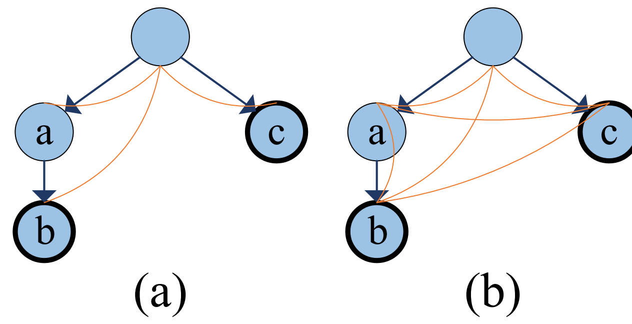


Figure 6.1: Exemples de liens avec la version actuelle (a), et avec la proposition future (b)

# RÉFÉRENCES

[1] R&M, «R&M Data Center Handbook,» Wetzikon, Switzerland, 2011.

[2] B. Mukherjee, L. Heberlein et K. Levitt, «Network intrusion detection,» *IEEE Network,* vol. 8, n° 3, pp. 26-41, May 1994.

[3] K. A. Scarfone et P. M. Mell, «SP 800-94. Guide to Intrusion Detection and Prevention Systems (IDPS),» National Institute of Standards & Technology, Gaithersburg, MD, United States, 2007.

[4] M. Roesch, «Snort - Lightweight Intrusion Detection for Networks,» chez *Proceedings of the 13th USENIX Conference on System Administration*, Berkeley, CA, USA, 1999.

[5] S. Anithakumari et D. Chithraprasad, «An Efficient Pattern Matching Algorithm for Intrusion Detection Systems,» chez *IEEE International Advance Computing Conference*, 2009.

[6] Snort. (2014) *Snort*. [En ligne]. Disponible : https://www.snort.org/

[7] M. Roesch, «SNORT Users Manual 2.9.7,» 2014.

[8] S. Kumar et E. H. Spafford, *An Application of Pattern Matching in Intrusion Detection,* 1994.

[9] Internet Live Stats. (2016) *Google Search Statistics*. [En ligne]. Disponible : http://www.internetlivestats.com/google-search-statistics/#trend

[10] U. Hölzle. (2009) *Powering a Google search*. [En ligne]. Disponible : https://googleblog.blogspot.ca/2009/01/powering-google-search.html

[11] J. Glanz. (2012) *Power, Pollution and the Internet*. [En ligne]. Disponible : http://nyti.ms/18nZqnG

[12] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal et S. Pope, «A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services,» chez *41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[13] I. King. (2015) *Intel's $16.7 Billion Altera Deal Is Fueled by Data Centers*. [En ligne]. Disponible : http://www.bloomberg.com/news/articles/2015-06-01/intel-buys-altera-for-16-7-billion-as-chip-deals-accelerate

[14] A. B. Lacroix, J. P. Langlois, F.-R. Boyer, A. Gosselin et G. Bois, «Node Configuration for the Aho-Corasick Algorithm in Intrusion Detection Systems,» chez *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, New York, NY, USA, 2016.

[15] J. E. Hopcroft, R. Motwani et J. D. Ullman, Introduction to Automata Theory, Languages, and Computation (2nd Edition), Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[16] A. V. Aho et M. J. Corasick, «Efficient String Matching: An Aid to Bibliographic Search,» *Commun. ACM,* vol. 18, n° 6, pp. 333-340, June 1975.

[17] I. Kuchir. *Aho-Corasick – implementation and animation.* [En ligne]. Disponible : http://blog.ivank.net/aho-corasick-algorithm-in-as3.html

[18] N. Tuck, T. Sherwood, B. Calder et G. Varghese, «Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection,» chez *IEEE Infocom, Hong Kong*, 2004.

[19] X. Zha et S. Sahni, «Highly compressed Aho-Corasick automata for efficient intrusion detection,» chez *IEEE Symposium on Computers and Communications (ISCC)*, 2008.

[20] A. Bremler-Barr, Y. Harchol et D. Hay, «Space-time tradeoffs in software-based deep Packet Inspection,» chez *IEEE 12th International Conference on High Performance Switching and Routing (HPSR)*, 2011.

[21] T. Nishimura, S. Fukamachi et T. Shinohara, «Speed-up of Aho-Corasick pattern matching machines by rearranging states,» chez *Eighth International Symposium on String Processing and Information Retrieval (SPIRE)*, 2001.

[22] V. Dimopoulos, I. Papaefstathiou et D. Pnevmatikatos, «A Memory-Efficient Reconfigurable Aho-Corasick FSM Implementation for Intrusion Detection Systems,» chez *2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2007.

[23] G. Shenoy, J. Tubella et A. Gonzalez, «A Performance and Area Efficient Architecture for Intrusion Detection Systems,» chez *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2011.

[24] N. P. Tran, M. Lee, S. Hong et M. Shin, «Memory Efficient Parallelization for Aho-Corasick Algorithm on a GPU,» chez *IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[25] Y. Xu, L. Ma, Z. Liu et H. Chao, «A Multi-dimensional Progressive Perfect Hashing for High-Speed String Matching,» chez *7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2011.

[26] L. Tan et T. Sherwood, «A High Throughput String Matching Architecture for Intrusion Detection and Prevention,» chez *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, Washington, DC, USA, 2005.

[27] H.-J. Jung, Z. K. Baker et V. K. Prasanna, «Performance of FPGA implementation of bit-split architecture for intrusion detection systems,» chez *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006.

[28] C. H. Lin, S. Y. Tsai, C. H. Liu, S. C. Chang et J. M. Shyu, «Accelerating String Matching Using Multi-Threaded Algorithm on GPU,» chez *IEEE Global Telecommunications Conference (GLOBECOM)*, 2010.

[29] Y. S. Dandass, S. C. Burgess, M. Lawrence et S. M. Bridges, «Accelerating String Set Matching in FPGA Hardware for Bioinformatics Research,» *BMC Bioinformatics,* vol. 9, n° 1, pp. 1-11, 2008.

[30] P. Piyachon et Y. Luo, «Efficient memory utilization on network processors for deep packet inspection,» chez *ACM/IEEE Symposium on Architecture for Networking and Communications systems*, 2006.

[31] P. Piyachon et Y. Luo, «Design of High Performance Pattern Matching Engine Through Compact Deterministic Finite Automata,» chez *Proceedings of the 45th Annual Design Automation Conference*, New York, NY, USA, 2008.

[32] R. Smith, C. Estan et S. Jha, «XFA: Faster Signature Matching with Extended Automata,» chez *IEEE Symposium on Security and Privacy*, 2008.

[33] X. Chen, K. Ge, Z. Chen et J. Li, «AC-Suffix-Tree: Buffer Free String Matching on Out-of-Sequence Packets,» chez *7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2011.

[34] C.-C. Chen et S.-D. Wang, «An Efficient Multicharacter Transition String-matching Engine Based on the Aho-corasick Algorithm,» *ACM Trans. Archit. Code Optim.,* vol. 10, n° 4, pp. 25:1--25:22, Dec 2013.

[35] S. Arudchutha, T. Nishanthy et R. G. Ragel, «String matching with multicore CPUs: Performing better with the Aho-Corasick algorithm,» chez *2013 IEEE 8th International Conference on Industrial and Information Systems*, 2013.

[36] A. Tumeo, O. Villa et D. Sciuto, «Efficient Pattern Matching on GPUs for Intrusion Detection Systems,» chez *Proceedings of the 7th ACM International Conference on Computing Frontiers*, New York, NY, USA, 2010.

[37] D. Pao, W. Lin et B. Liu, «Pipelined Architecture for Multi-String Matching,» *IEEE Computer Architecture Letters,* vol. 7, n° 2, pp. 33-36, July 2008.

[38] K. Kanani. (2013) *Aho-Corasick implementation (part of multifast)*. [En ligne]. Disponible : http://multifast.sourceforge.net/

[39] Netresec. (2012) *Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC)*. [En ligne]. Disponible : http://www.netresec.com/?page=MACCDC

[40] K. A. Scarfone et P. M. Mell, «SP 800-94. Guide to Intrusion Detection and Prevention Systems (IDPS),» National Institute of Standards and Technology, Gaithersburg, MD, United States, 2007.