# Chaînage automatisé de programmes *BPF* par application de correctifs

par

Alexis BRODEUR

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE
M. Sc. A.

MONTRÉAL, LE 26 SEPTEMBRE 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

**PRÉSENTATION DU JURY**

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Abdelouahed Gherbi, directeur de mémoire
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Alain April, président du jury
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Kaiwen Zhang, membre du jury
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

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

LE 8 SEPTEMBRE 2023

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

## REMERCIEMENTS

Je tiens à remercier mon professeur Abdelouahed Gherbi, qui m'a constamment guidé et motivé dans mes études au baccalauréat et à la maîtrise. Je tiens aussi à remercier ma famille, mes collègues et mes amis pour leurs encouragements lors de mon parcours universitaire.

**Chaînage automatisé de programmes *BPF* par application de correctifs**

Alexis BRODEUR

## RÉSUMÉ

*BPF* est une technologie du noyau de *Linux* recevant une augmentation fulgurante d'attention dans les domaines de la télécommunication et de la sécurité informatique. *BPF* permet l'exécution de programmes directement dans le noyau de *Linux* pour de grandes améliorations de performances. Ces programmes peuvent observer et surveiller le système, filtrer des paquets réseau, router des paquets, etc. Dans bien des cas d'utilisation, il est nécessaire d'appliquer plusieurs opérations orthogonales séquentiellement dans des programmes *BPF*. Permettre l'enchaînage automatisé de programmes implémentant ces opérations permet d'ouvrir la porte à un écosystème plus vif et dynamique permettant la composition de programmes. En permettant la consommation de programmes *BPF* orthogonaux, il devient intéressant de permettre aux programmes d'être utilisés de la même façon que nous utilisons les conteneurs aujourd'hui. Avec une telle approche, composer plusieurs programmes provenant de sources tiers afin d'obtenir un comportement désiré offre des avantages considérables pour l'industrie. Nous proposons donc dans ce travail une nouvelle approche pour enchaîner des programmes *BPF* qui ne présente pas de perte de performances significative, et présentant une meilleure compatibilité sur diverses architectures *CPU* et versions du noyau de *Linux*. Nous comparons cette approche à d'autres approches existantes telles *XDP Dispatcher* et le chaînage manuel. Nous comparons la performance de notres approches avec les autres par l'évaluation du débit maximum, temps *CPU*, et nombre de cycles *CPU* de chaque approches pour toutes les versions *LTS* de *Linux* présentement supportée lors de l'écriture de ce travail. Nous montrerons par ces résultats que notre approche ne souffre pas de pertes de performances significatives comparativement aux autres approches, et nous argumentons de points de comparaison plus subjectifs tel la compatibilité, l'orthogonalité des programmes, etc. Il est espéré que ce travail aidera à l'implémentation de nouveaux outils pour l'écosystème *BPF*.

**Mots-clés:** linux, kernel, bpf

**Automated sequencing of BPF programs by patching**

Alexis BRODEUR

## ABSTRACT

BPF is an important technology at the core of the Linux kernel that has received an increase in attention and contribution from multiple fields of computer science like networking and security. BPF allows execution of precompiled programs directly inside the Linux kernel for significant performance improvements over other approaches. These programs can monitor a running system, filter network packets, route network packets, etc. In many use cases, there is a need to run multiple orthogonal programs sequentially, like packet filtering or packet routing. Allowing the automation of program chaining will allow even more notable improvements to the ecosystem surrounding this technology by improving how programs can be coupled together to acheive more complex use cases. In this work, we propose a novel approach to chain the execution of BPF programs without harming performance and offering better compatibility compared to similar approaches. We compare our novel approach with XDP Dispatcher and manual program chaining. For all approaches, we compare performances by measuring the transfer speed, run time and CPU cycles used for multiple LTS versions of the Linux kernel. We show that our approach is a viable alternative that does not suffer from performance costs and argue its boons on more subjective matters like compatibility, program orthogonality, etc. We hope this work will aid in the development of new tools in the BPF ecosystem.

**Keywords:** linux, kernel, bpf

## TABLE DES MATIÈRES

Page

## LISTE DES TABLEAUX

Page

## LISTE DES FIGURES

## LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

| | |
|---|---|
| BPF | *Berkely Packet Filter* |
| XDP | *eXpress Data Path* |
| BSD | *Berkeley Software Distribution* |
| OSI | *Open System Interconnect* |
| API | *Application Programming Interface* |
| cBPF | *classical Berkeley Packet Filter* |
| eBPF | *extended Berkeley Packet Filter* |
| JIT | *Just In Time* |
| RISC | *Reduced Instruction Set Computer* |
| BCC | *BPF Compiler Collection* |
| ELF | *Executable and Linkable Format* |
| BTF | *BPF Type Format* |
| CPU | *Central Processing Unit* |
| IPv6 | *Internet Protocol version 6* |
| TCP | *Transmission Control Protocl* |
| LTS | *Long Term Support* |

## LISTE DES SYMBOLES ET UNITÉS DE MESURE

| | |
|---|---|
| b | Bit |
| O | Octet |

# INTRODUCTION

Ces dernières années, *Berkeley Packet Filter* (*BPF*) est une technologie du noyau *Linux* ayant vue un renouveau fulgurant. Depuis 2014, plusieurs changements majeurs ont été apportés à son implémentation dans le noyau *Linux*, ces changements apportant de grandes augmentations en performance ainsi qu'une plus haute versatilité. Aujourd'hui, *BPF* est une technologie polyvalente dans les domaines de la réseautique et de la télémesure, expliquent Calavera & Fontana (2019). Selon plusieurs experts dans le domaine de la télécommunication, incluant Bernier (2021), *BPF* présente des difficultés de développement, mais aussi des avantages considérables permettant une amélioration de performances majeure dans les réseaux de télécommunications modernes.

## Contexte

Nous considérons que *BPF* est une technologie très intéressante pour les domaines de la réseautique et de télécommunication. Comme mentionné par Bernier, l'amélioration des performances comparativement à des approches basées sur la pile réseau traditionnelle présente des occasions intéressantes. *BPF* présente aussi des circonstances opportunes intéressantes en sécurité et l'observabilité dut à son habileté d'être attaché à des évènements autres que les paquets réseau, tels des appels système, des appels de fonctions du noyau de *Linux*, des appels de fonctions de logiciels utilisateurs, etc.

## Problématique

*BPF* est une technologie assez récente qui commence à attirer l'intérêt dans le secteur des entreprises et académiques. Par notre travail avec cet outil, nous déterminons que le multiplexage de programmes est un problème majeur à survenir pour aider *BPF* à être plus facilement intégrable dans des solutions logicielles. Actuellement, il est impossible pour des programmes *BPF* d'être multiplexé sur un nombre considérable d'évènements, car ces derniers n'acceptent

que l'attache d'un programme *BPF* individuel. Nous considérons cette problématique très importante à résoudre afin de permettre la création d'outils et d'applications intéressantes pour *BPF* tels des cadriciels de chargement de programmes *BPF*, des *Service Function Chaining* (*SFC*) de *Network Function Virtualization* (*NFV*) en *BPF*, etc.

**Objectifs**

Pour permettre la création d'applications et d'outils mentionnés précédemment et bien d'autres, nous présentons donc une approche nouvelle. Cette approche doit être en mesure de multiplexer plusieurs programmes *BPF* orthogonaux sans nécessiter la modification préalable des programmes pour les rendre compatibles avec notre approche. Notre approche doit aussi présenter des caractéristiques de performances équivalentes aux autres approches auxquelles nous allons la comparer. Notre approche doit aussi présenter une meilleure compatibilité pour différentes architectures de processeurs et de versions du noyau de *Linux*.

**Contribution**

Notre approche est basée sur la création d'une chaîne de programmes *BPF*. Les programmes *BPF* sont modifiés avant d'être chargés dans le noyau de *Linux* en appliquant un correctif avant les instructions du programme pour inclure un préambule permettant la redirection par un appel de queue au prochain programme dans la chaîne. Ce correctif, appliqué avant les instructions des programmes, est aussi appliqué à l'information *BPF Type Format* (*BTF*) afin de préserver l'information de relocalisation lorsque les programmes sont chargés par des outils tels *libbpf*. Notre approche présente donc une méthode permettant très facilement la composition de programmes. Dans le cadre de *SFC* de *NFV* en *BPF*, il devient possible de composer plusieurs *NFV* sans avoir à modifier le code source des *NFV* afin d'obtenir un comportement spécifique aux besoins souhaités.

Notre champ de recherche étant relié à l'observabilité et la surveillance de systèmes par l'utilisation de *BPF*, la composition de programmes est une de nos contributions majeures. Dans le cadre de l'observabilité et la surveillance, il arrive fréquemment que nous ayons à attacher des programmes *BPF* à des endroits dans le noyau de *Linux* ne permettant pas le multiplexage. Notre contribution, qui comporte de multiples avantages, nous permettra précisément à multiplexer l'exécution de nos programmes de surveillance avec d'autres programmes *BPF* traditionnels. C'est dans cette optique que nous présentons cette contribution, et d'autres telles Brodeur, Boukhtouta, Necir Medakene & Gherbi (2023).

**Sommaire**

Dans le chapitre 1, nous commençons par expliquer des concepts fondamentaux nécessaires pour la compréhension du reste du mémoire. Par la suite dans le chapitre 2, nous faisons une revue de la littérature dans le cadre de la rédaction de notre mémoire. Cette littérature peut servir à introduire des concepts fondamentaux plus avancés reliés, des solutions différentes à la problématique pour laquelle nous soumettons ce travail, ou pour introduire des points de discussions. Le chapitre 3 discute ensuite de notre approche, la contribution à la problématique que nous souhaitons mettre en évidence par ce travail. Le chapitre 4 discute par la suite de la méthodologie que nous avons utilisée pour comparer les performances entre notre approche et les approches existantes aujourd'hui. Le chapitre 5 présente les résultats obtenus lors de nos expériences et offre des explications sommaires sur les résultats. Le chapitre 5 va aussi présenter les résultats des expériences pour discuter de notre approche objectivement et subjectivement. Nous allons finalement conclure ce mémoire par une conclusion récapitulant le travail accompli et présentant nos recommandations.

# CHAPITRE 1

# CONCEPTS FONDAMENTAUX

## 1.1 Linux

*Linux* est un noyau de système d'exploitation *FOSS* (*Free and Open Source Software* créé par Linus Torvalds comme solution de rechange à *UNIX* dans une ère ou *AT&T* et *Berkeley Software Distribution* (*BSD*) rencontrait des problèmes de licenciement et de commercialisation de système d'exploitation *UNIX-like* selon Raymond (2003).

Aujourd'hui, *Linux* et les distributions l'utilisant forment un écosystème robuste de systèmes d'exploitation. *Linux* et ses dérivés sont très fréquemment utilisés dans les centres de serveurs, ainsi qu'avec des technologies liées à la visualisation.

*Linus* étant un système logiciel d'envergure, cette section présente les parties de *Linus* importantes pour comprendre le reste de ce travail.

### 1.1.1 Pile réseau

Comme documenté par Benvenuti (2006), les bases de la pile réseau de *Linux* reposent sur 3 structures de données essentielles, soit *socket*, *sock*, et *sk_buff*. ***socket*** est l'abstraction avec laquelle les développeurs de programmes d'applications réseau font affaire. Son implémentation suit la spécification des *Berkeley Socket Interface*, une spécification, les méthodes d'interactions entre les programmes d'un système d'exploitation et les *socket*. ***sock*** est la représentation interne du noyau de *Linux* des interfaces réseau configurées. ***sk_buff*** est la représentation interne du noyau de *Linux* des paquets réseau entrant et sortant des *sock*.

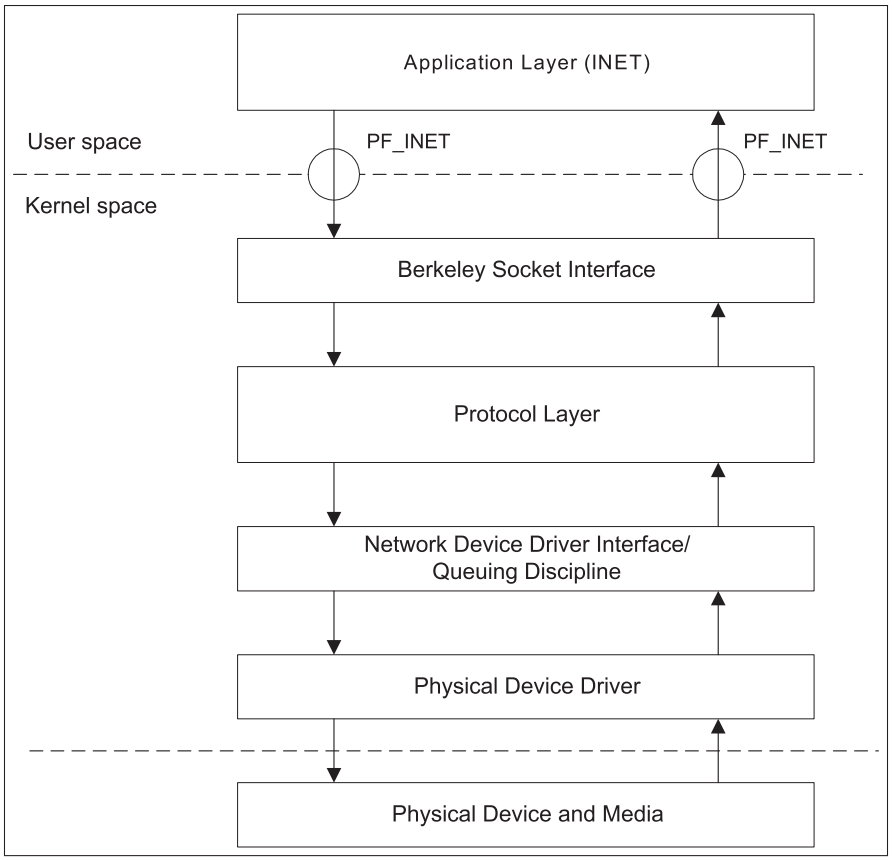


Figure 1.1 Architecture réseau de Linux, Kasatkin (2001)

La figure 1.1 présente un aperçu à haut niveau de la pile réseau de *Linux*. La pile réseau inclue les pilotes réseaux, l'ordonnancement des paquets réseaux, l'encodage et le décodage d'entêtes de paquets au niveau de la couche réseau et transport du modèle *Open System Interconnect* (*OSI*), ainsi que la *Berkeley Socket Interface* pour communiquer avec les programmes utilisateurs.

### 1.1.2 Perf Events

*perf_event* est une *Application Programming Interface* (*API*) du noyau de *Linux* permettant d'obtenir de l'information du système en cours d'exécution. *perf_event* a principalement été créé pour analyser les problèmes de performances dans le noyau et les applications de *Linux*. *perf_event* est utilisé pour obtenir de l'information de plusieurs évènements tels le nombre de cycles du processeur, l'exécution de *kprobe* ou de *uprobe*, des *tracepoints* dans le noyau, etc.

### 1.1.3 Kprobe et Uprobe

Keniston, Panchamukhi & Hiramatsu (2022) documentent dans les pages manuelles de *Linux* l'utilité des *kprobe* et *uprobe* dans l'introspection de systèmes. Les *kprobe* et *uprobe* permettent respectivement d'obtenir de l'information lors de l'exécution de code du noyau et de code d'applications utilisateurs sur *Linux*.

Keniston *et al.* expliquent aussi le fonctionnement relativement simple des *kprobes* et *uprobes*. Lorsqu'un *kprobe* et *uprobe* est enregistré auprès du noyau par l'*API* des *perf_event*, une instruction de la fonction à superviser est remplacée par une instruction causant un point d'arrêt auprès du processeur (e.x. : *int3* sur *x86*). Le noyau détecte ce point d'arrêt pour exécuter le *kprobe* ou *uprobe*, rejoue l'instruction originale, puis reprend l'exécution de la fonction.

## 1.2 Berkeley Packet Filter

Originalement, plusieurs versions de *UNIX* offraient des fonctionnalités de filtrage de paquets réseau. Cela allait de *Network Interface Tap* pour *SunOS* Sun Microsystems Inc. (1990) à *Ultrix Packet Filter* pour le système *Ultrix* de *DEC* Mogul, Rachid & Accetta (1987), etc. En 1992, McCanne et Jacobson proposent un nouveau filtreur de paquets réseau, inspiré par le travail de *Mogul et al.* dans *Ultrix*. Ce nouveau filtreur de paquet prend en compte la consolidation des processeurs vers l'architecture *x86* de *Intel* Mogul *et al.* (1987). Ce filtreur de paquets réseaux, intitulé *BPF* (*Berkeley Packet Filter*) fût intégré au noyau de *Linux* et vît une forte adoption dans divers outils au travers de *libpcap* tels *WireShark* et *tcpdump*.

### 1.2.1 Classical Berkeley Packet Filter

Dans la champ lexical moderne entourant *Linux* et son écosystème, *classical Berkeley Packet Filter* (*cBPF*) réfère à l'implémentation originale proposée en 1992 par McCanne et Jacobson. Le reste de cette section reprend donc l'information présentée par *Mogul et al. 1987* sauf indication contraire.

*cBPF* fut introduit par McCanne et Jacobson comme ajout au noyau de *Linux* offrant un jeu d'instructions pour filtrer des paquets réseau. Ce jeu d'instructions se caractérise par des instructions de taille fixe de 64 bits permettant des opérations sur des registres de 32 bits. Le tableau 1.1 présente l'encodage d'une instruction *cBPF*.

Tableau 1.1 Encodage d'une instruction *cBPF*

| Champ | Position (Bits) | Taille (Bits) | Description |
|---|---|---|---|
| code | 0 | 16 | Le code de l'instruction *BPF* à exécuter. |
| jt | 16 | 24 | Une position relative vers laquelle sauter si l'instruction est un saut, et que la condition du saut est vraie. |
| jf | 24 | 32 | Une position relative vers laquelle sauter si l'instruction est un saut, et que la condition du saut est fausse. |
| k | 32 | 64 | Valeur immédiate à l'usage. |

Pour des raisons de simplicité, *cBPF* offre l'accès à seulement 2 registres aux utilisateurs. Un troisième registre existe contenant l'information le *frame pointer* du programme en cours d'exécution, mais ce troisième registre n'est pas accessible par les utilisateurs de *cBPF*. Le tableau 1.2 ci-dessous présente l'information des registres disponibles aux utilisateurs :

Tableau 1.2 Registres utilisables dans *cBPF*

| Mnémonique | Description |
|---|---|
| X | Un registre universel de travail. |
| A | L'accumulateur pour les instructions arithmétiques. |

Les registres sont une forme de mémoire, mais ne sont pas la seule mémoire auquel les programmes *cBPF* ont accès. Comme mentionné plus haut, un tableau en mémoire *M* était disponible pour persister de l'information pour la durée de vie du programme. Le programme

*cBPF* est aussi en mesure d'adresser directement le paquet réseau à filtrer. La tableau 1.3 présente les différents modes d'adressages supportés par diverses instructions *cBPF*.

Tableau 1.3 Modes d'adressages en *cBPF*

| Mode | Description |
|---|---|
| 0 | Direct par le registre $X$. |
| 1 | Indirect par valeur immédiate dans le paquet réseau. |
| 2 | Indirect par le registre $X$ en fonction d'un offset dans le paquet réseau. |
| 3 | Indirect par valeur immédiate dans la région $M$ de mémoire. |
| 4 | Direct par valeur immédiate. |
| 5 | Indirect par valeur immédiate dans le premier octet du paquet réseau. |
| 6 | Direct par adresse de l'instruction au libellé $L$. |
| 7 | Saut vers $Lt$ si la valeur immédiate est vrai, sinon sautes vers $Lf$. |
| 8 | Saut vers $Lt$ si le registre $X$ est vrai, sinon sautes vers $Lf$. |
| 9 | Saut vers $Lt$ si la valeur immédiate est vrai. |
| 10 | Saut vers $Lt$ si le registre $X$ est vrai. |
| 11 | Direct par le registre $A$. |
| 12 | Extension *BPF* |

### 1.2.2 Extended Berkeley Packet Filter

En 2014, Alexei Starovoitov, un employé de *PlumGrid*, commence à travailler sur *extended Berkeley Packet Filter* (*eBPF*). Les premières versions de *eBPF* sont rejetées du noyau de *Linux* à cause de l'envergure des changements. Starovoitov comprend donc qu'il doit faire migrer *cBPF* part partit vers la version de *eBPF* envisagée. Les premières versions de *eBPF* ne sont donc que de changements internes au format d'exécution, ainsi que d'un moteur de compilation *Just In Time* (*JIT*) pour ce nouveau format interne intitulé *internal BPF*. Au fil des mois, et avec plus de fonctionnalités ajoutées, Starovoitov (2022) fait le point auprès d'autres développeurs du noyau *Linux* pour que *internal BPF* soit exposé aux utilisateurs de *Linux* sous le nom de *eBPF*.

Comparativement à *cBPF*, qui ne servait qu'à filtrer des paquets réseaux, justifier exposer *eBPF* aux utilisateurs de *Linux* a nécessité un pivot vers différentes fonctionnalités impossibles à accomplir en *cBPF*. *eBPF* pivote donc rapidement pour supporter l'exécution lors du déclenchement de *perf_event* et d'appels de fonctions dans le noyau de *Linux*. Aujourd'hui, *eBPF* peut être attaché sur une multitude de déclencheurs d'événements, appelés des points d'attache.

*eBPF* a été créer pour avoir une forte ressemblance aux jeux d'instructions d'ordinateurs *Reduced Instruction Set Computer* (*RISC*). En ayant un jeu d'instructions beaucoup plus proche de ce qui est réellement disponible sur les ordinateurs physiques permet à *eBPF* d'être compilé en un code machine très performant, bien plus que *cBPF* pourrait l'être. Starovoitov (2022) explique que l'ancien encodage des instructions sous *cBPF* avec 2 adresses de saut (voir la figure 1.1) empêche l'implémentation de *jump-threading* par le compilateur *JIT* du noyau de *Linux*.

*eBPF* est un jeu d'instructions pour le noyau de *Linux* ayant comme base le jeu d'instructions *cBPF* retravaillé pour avoir accès à plus de registres et des registres de 64 bits. Pour être facilement rétrocompatible avec *cBPF*, la taille des instructions reste inchangée afin de pouvoir facilement traduire les instructions *cBPF* en instructions *eBPF* sans avoir à allouer de mémoire. Le tableau 1.4 décrit l'encodage des instructions *eBPF* qui sont très similaires à celui des instructions *cBPF* :

Tableau 1.4 Encodage d'une instruction *eBPF*

| Champ | Position (Bits) | Taille (Bits) | Description |
|---|---|---|---|
| opcode | 0 | 8 | Le code de l'instruction *BPF* à exécuter. |
| src_reg | 8 | 4 | Le registre source associé à l'instruction. |
| dst_reg | 12 | 16 | Le registre destinataire associé à l'instruction. |
| offset | 16 | 32 | *Offset* utilisé pour l'arithmétique de pointeurs. |
| imm | 32 | 64 | Valeur immédiate sous format d'un entier signé. |

L'*opcode* d'une instruction *eBPF* est-elle aussi subdivisée en plusieurs sous-champs encodant encore plus d'information pertinente concernant l'instruction. Le tableau 1.5 explique cet encodage.

Tableau 1.5 Encodage du code d'une instruction *eBPF*

| Nom | Position (bits) | Taille (Bits) | Description |
|---|---|---|---|
| instruction_class | 0 | 3 | La classe d'instruction. |
| source | 3 | 1 | La provenance de l'opérande source, soit vrai pour $src\_reg$ ou faux pour $imm$. |
| code | 4 | 4 | Le code de l'opération à exécuter. |

La classe de l'instruction représente à haut niveau la catégorie de l'instruction, tels une opération arithmétique, un saut, etc. Le code contient, en fonction de la classe d'instruction, le numéro de l'instruction exacte à exécuter.

Tableau 1.6 Registres utilisables dans *eBPF*

| Mnémonique | Description |
|---|---|
| r0 | Registre contenant le valeur de retour des fonctions |
| r1 | Registre contenant le premier argument d'un appel de fonction |
| r2 | Registre contenant le second argument d'un appel de fonction |
| r3 | Registre contenant le troisième argument d'un appel de fonction |
| r4 | Registre contenant le quatrième argument d'un appel de fonction |
| r5 | Registre contenant le cinquième argument d'un appel de fonction |
| r6 | Registre enregistré et rétabli par la fonction appelée |
| r7 | Registre enregistré et rétabli par la fonction appelée |
| r8 | Registre enregistré et rétabli par la fonction appelée |
| r9 | Registre enregistré et rétabli par la fonction appelée |
| r10 | Pointeur vers le *frame pointer* pour la pile de la fonction |

*eBPF* étant un jeu d'instruction voulant être plus moderne que *cBPF* et plus en ligne en les architectures de processeurs modernes, une augmentation considérable des registres disponibles

comparativement à *cBPF* est disponible. Comme le présente le tableau 1.6, *eBPF* dispose maintenant de 10 registres.

### 1.2.3 Maps

Les programmes *BPF* s'exécutant dans le noyau de *Linux* avec un jeu d'instructions très simples, un mécanisme simple intitulé les *maps* existe pour permettre les structures de données de taille dynamique et la communication entre les programmes *BPF* dans le noyau et les applications utilisant ces programmes *BPF*.

Tableau 1.7 Énumération non-exhaustive des *maps* en *BPF*

| Nom | Description |
|---|---|
| BPF_MAP_TYPE_ARRAY | Un tableau |
| BPF_MAP_TYPE_HASH | Un *hash* associatif |
| BPF_MAP_TYPE_PER_CPU_ARRAY | Un tableau où chaque *CPU* a sa propre copie |
| BPF_MAP_TYPE_ARRAY_OF_PROGS | Un tableau de programmes |
| BPF_MAP_TYPE_HASH_OF_MAPS | Un hash de *maps* |

Le tableau 1.7 présente quelques sortes de *maps* disponibles en *BPF*. Toute création de *maps* requiert un type, une taille de clef, une taille de valeur, ainsi qu'un nombre maximum d'entrées dans le tableau. Des fonctions auxiliaires sont offertes aux programmes *BPF* pour lire et écrire dans les *maps*, et l'application gérant les *maps* peut elle aussi lire et écrire dedans à d'un appel système.

### 1.2.4 Vérificateur

*BPF* étant un jeu d'instructions exécuté directement dans le noyau de *Linux*, une attention particulière doit être apportée à la sécurité d'un tel jeu d'instruction. *BPF* offre un jeu

d'instructions non privilégié, et pour qu'un programme soit chargé avec succès dans le noyau de *Linux*, il doit être accepté par un vérificateur strict Kernel development community (2023b).

Selon la documentation du noyau de *Linux*, le vérificateur fait son travail en deux étapes. La première étape utilise un graphe acyclique directionnel pour empêcher les boucles. La deuxième étape utilise un graphe de flot de contrôle pour d'autres validations, telles des instructions non exécutées.

Le vérificateur de programmes *BPF* empêche les programmes d'effectuer toutes opérations posant des risques de sécurité dans le noyau de *Linux*. Pour cette raison, il est souvent probable qu'un programme sécuritaire valide soit rejeté par le vérificateur.

### 1.2.5 Cycle de développement

De notre expérience personnelle, il existe présentement 2 outils utilisés lors du développement de programmes *BPF*. Ces 2 outils, *libbpf* et *BCC*, sont les principaux outils utilisés dans le cadre de développement de logiciels faisant l'utilisation de programmes *BPF*.

*BPF Compiler Collection* (*BCC*) est un cadriciel entourant *LLVM* et l'*API BPF* du noyau de *Linux* afin d'offrir un cycle de développement rapide. Ce cadriciel supporte présentement les langages de programmation *Lua* et *Python* BCC development community (2023). Avec l'utilisation de *BCC*, le code source des programmes *BPF* fait partit des livrables, et ces programmes sont compilés à la dernière minute sur la machine exécutant le code *Python* ou *Lua*. Cela implique aussi une dépendance directe en production sur *LLVM*.

*libbpf* est une librairie par les développeurs du noyau de *Linux*. Contrairement à *BCC*, *libbpf* fournit un *API* en *C*. Les programmes *BPF* sont distribués sous forme de fichiers respectant le *Excutable and Linkable Format* (*ELF*) incorporés dans l'application les utilisant. Avec l'aide de *BPF Type Format* (*BTF*), *libbpf* offre des fonctionnalités suivant le principe *Compile Once, Run Everywhere* (*CO-RE*) afin de rendre la distribution de programmes très facile sur différents ordinateurs libbpf Authors (2023).

## 1.3 Executable and Linkable Format

*Executable and Linkable Format* (*ELF*) est un format ouvert utilisé par plusieurs systèmes d'exploitation basés sur les philosophies de *UNIX*, tel *Linux*. Ce format permet la distribution d'applications et de librairies partagées. Bien que plusieurs formats existent pour répondre à ce besoin, la communauté des systèmes d'exploitation *UNIX* s'est regroupée vers l'utilisation de *ELF* Messier (2015).

*libbpf* fait une grande utilisation du format de fichier *ELF* afin de regrouper les instructions *BPF* avec leurs métadonnées, telles les *maps* utilisées, et l'information *BTF* associée aux programmes contenus dans les fichiers *ELF*. Chaque programme est contenu dans une section du fichier *ELF* éponyme au point d'accrochage du programme. Un programme *BPF* pour l'entrée de l'appel système «openat» est donc contenu dans une section «syscalls/sys_entry_openat».

## 1.4 BPF Type Format

*BPF Type Format* (*BTF*) est un format de métadonnées créé pour encoder l'information de *maps BPF*. Au fil du temps, *BTF* a été étendu pour aussi inclure de l'information de débogage, ainsi que de typage de fonctions et structure de données Kernel development community (2023a).

*BTF* est principalement utilisé par *libbpf*. Lors de la distribution de programmes *BPF* avec *libbpf*, les programmes *BPF* sont incorporés dans l'application les utilisant sous un tableau d'octet représentant les données d'un fichier *ELF*. *BTF* utilise quelques sections dans ce fichier *ELF* pour décrire les *maps* utilisées par les programmes *BPF*, de l'information de débogage, ainsi que les correctifs, que *libbpf* doit appliquer aux programmes pour qu'ils pouvoir appliquer des techniques *CO-RE*.

# CHAPITRE 2

# REVUE DE LITTÉRATURE

Dans cette section nous révisons de la littérature explorant le chaînage de programmes *BPF*. Nous commençons cependant par introduire *XDPh* (*eXpress Data Path*), une technologie et cadriciel clef pour notre recherche, puisqu'une majeure partie de la littérature ne fonctionne qu'avec les programmes *BPF* de type *XDP*.

## 2.1 Express Data Path

*XDP* (*eXpress Data Path*) est un cadriciel offrant un nouveau type de programme *BPF* ainsi qu'un nouveau point d'attache du même nom introduit par Høiland-Jørgensen *et al.* (2018). Le point d'attache *XDP* se situe à l'entrée des paquets réseau dans la pile réseau de *Linux*. Le principal avantage de *XDP* comparativement aux points d'attache sur les *socket* est la performance que peut atteindre *XDP*, car les programmes *XDP* sont exécutés avant la pile réseau de *Linux*.

Comme cadriciel, *XDP* réutilise l'implémentation de *BPF* du noyau de *Linux* comme implémentation générique, et incorpore un nouveau point d'attache pour les programmes *XDP*. Avec ce nouveau point d'attache, multiples fonctions utilitaires sont aussi offertes à ces programmes. *XDP* offre spécifiquement des fonctions utilitaires pour permettre la manipulation de paquets réseau, comme des fonctions pour agrandir ou rapetisser la mémoire au début ou à la fin du paquet.

Les programmes *XPD*, comparativement aux autres types de programmes *BPF* sont exécutés aux alentours du pilote des cartes réseau. Tel que le montre la figure 1.1, cela implique que le traitement de paquets par *XDP* ne nécessite pas le passage par la pile réseau de *Linux*, permettant des gains majeurs de performance. Les cartes réseau modernes étant aussi de plus en plus puissantes, certaines de ces cartes réseau permettent aussi l'exécution de programmes *XDP* dans un processus intitulé *offloading*, dans lequel les programmes *XDP* sont exécutés indépendamment du processeur du système hôte à même les cartes réseau compatibles.

Selon Bernier (2021), un expert du domaine de la télécommunication, *XDP* est une technologie clef afin d'améliorer les performances des réseaux de télécommunication, *XDP* étant souvent capable d'augmenter les performances d'un ordre complet de magnitude.

## 2.2 A Framework for eBPF-Based Network Functions in an Era of Microservices

Dans cet article, Miano, Risso, Bernal, Bertrone & Lu (2021) présente une approche intéressante pour créer des chaînes de fonctions *BPF* dans le but d'implémenter des chaînes de *Network Function*s. Leur approche, basée sur un cadriciel de compilation et chargement de logiciels intitulés *BCC* (*BPF Compiler Collection*), leur permet de créer des chaînes de programmes en générant du code supplémentaire avant la compilation de programmes.

Bien que fonctionnelles, nous jugeons leur approche problématique, car elle est fortement dépendante sur *BCC*. Dans la communauté *Linux*, *BCC* commence à perdre en popularité pour être remplacé par l'outil *libbpf* offert par les concepteurs du noyau *Linux*. Contrairement à *libbpf*, la distribution de logiciels utilisant *BCC* nécessite aussi la distribution d'un compilateur ainsi que d'un lieur *C* complet, car *BCC* effectue de la génération de code lors de l'exécution de l'application.

Avec leur approche, *Polycube*, Miano *et al.* doivent donc faire l'analyse grammaticale de programmes en *C* pour préfixer un préambule. Cela implique que leur approche ne fonctionnerait pas en dehors de *BCC* et du langage de programmation *C*.

Miano *et al.* mentionnent que leur approche ne prend pas avantage de *BPF CO-RE*, et que cela ne les dérange pas. Ils argumentent que, effectivement, les programmes *TC* et *XDP* ont une *ABI* très stable, donc il n'est pas nécessaire d'utiliser *BPF CO-RE*. Nous pensons cependant que lorsque vient le temps de développer une solution permettant le chaînage de programmes *BPF* générique, offrir du support *BPF CO-RE* est primordial, car *TC* et *XDP* ne sont plus nos seules préoccupations.

## 2.3 eBPF : A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)

Soldani *et al.* (2023) présentent l'utilité de *BPF* dans les réseaux de télécommunications 5G. Ils mettent le point sur plusieurs avantages de *BPF* comme technologie clef dans le développement de nouvelles applications. Soldani *et al.* itèrent sur plusieurs notions intéressantes de *BPF* et leurs utilités dans les réseaux de télécommunication 5G, telle la sécurité des réseaux, l'interface avec des outils tel *Kubernetes*, ainsi que la monétisation de *BPF*.

Dans l'article, Soldani *et al.* présentent leur plateforme *BPF*, *Sauron*, un cadriciel permettant la gestion et facilitant le développement de programmes *BPF*. Leur cadriciel permet d'aider la conception et implémentation de modules entourant la surveillance réseau, la sécurité des applications, ainsi que la surveillance de la dissipation de la puissance. Avec ces genres de programmes, Soldani *et al.* sont en mesure de récolter plusieurs métriques clefs afin de comprendre le fonctionnement de réseaux 5G.

Bien que l'article existe pour présenter leur plateforme *Sauron*, Soldani *et al.* posent sur papier une présentation très complète des avantages de *BPF* dans le domaine de la réseautique et de la télécommunication. Ils présentent un bon aperçu des avantages que présente *BPF* et de l'orientation vers laquelle ces avantages pourraient entraîner la recherche.

## 2.4 Building an Extensible Open vSwitch Datapath

*BPF* offre des limites strictes d'exécution de programmes à cause du vérificateur faisant partie du noyau de *Linux*. Avec une limite de 1 million d'instructions par programmes et un vérificateur très stricte, il arrive souvent que des programmes *BPF* plus complexes soient refusés par le vérificateur.

Dans leur article, Tu, Stringer & Pettit (2017) offrent une solution pour construire un *OpenV Switch Datapath* comme une séquence de programmes *BPF*. Leur approche utilise un générateur

traduisant des fichiers *P4*, un langage de programmation de routage de paquets réseau, en code *BPF*.

Dans leur solution, chaque programme *BPF* est doit être informé de l'existence de la chaîne et du prochain programme dans la chaîne, impliquant qu'un changement à la chaîne nécessite une régénération des programmes *BPF* en fonction du code source *P4*. Dans le cadre de leur recherche, l'utilisation d'un générateur traduisant les programmes *P4* en programmes *BPF* atténue la complexité supplémentaire d'avoir à utiliser une chaîne de programmes explicites dans le code *BPF*.

Tu *et al.* (2017) montrent qu'il est possible de créer des chaînes de programmes manuellement, mais leur approche n'est pas extensible à d'autres domaines. Ils ne permettent pas le chaînage de programmes *BPF* arbitraires, et tous les programmes *BPF* de la chaîne ont connaissance les uns des autres.

## 2.5 XDP Dispatcher

*XDP Dispatcher* est l'implémentation de multiplexage provenant de *libxdp*, une bibliothèque de code permettant exclusivement le multiplexage de programmes sur les points d'attache *XDP* implémentée et documentée Høiland-Jørgensen & Kartikeya Dwivedi (2023).

*XDP Dispatcher* utilise une fonctionnalité du noyau de *Linux* intitulé *freplace*. *freplace* permet le chargement de programmes *BPF* à titre de remplacement de fonctions dans d'autres programmes. *libxdp* crée un programme *XDP*, le *dispatcher*, avec des fonctions vides servant de points d'attache par *freplace*. Les programmes écrits pour *libxdp* doivent déclarer dans une structure de données des métadonnées pour aider *libxdp* à comprendre comment les charger. *libxdp* charge un programme en remplaçant une fonction vide du *dispatcher* avec le code du programme à charger par l'utilisation de *freplace*.

*libxdp* permet ainsi le multiplexage de programmes *XDP*. Cependant, ce multiplexage présente quelques problèmes. *libxdp* ne garantit pas l'ordre d'exécution, bien que celui-ci puisse être configuré avec une priorité dans les métadonnées.

# CHAPITRE 3

# CHAÎNE DE PROGRAMMES BPF

*BPF* voit une explosion fulgurante en popularité dans plusieurs domaines des technologies de l'information et du génie logiciel. *BPF* est une technologie très intéressante dans plusieurs domaines, telles la sécurité, la réseautique, etc. Selon les experts du domaine, *BPF* est en voit de changer drastiquement le milieu de la télécommunication .

Une limitation majeure de *BPF* est l'inhabilité d'attacher plusieurs programmes types *BPF* sur le même point d'accrochage. Cette limitation présente un défi majeur, car les points d'accrochage les plus susceptibles de nécessiter l'exécution de plusieurs programmes, tels les programmes *XDP*, sont aussi ceux étant impactés par cette limitation. De plus, bien d'autres types de programmes supportent le multiplexage, il n'y a pas de garantie sur l'ordre d'exécution des programmes.

Dans cette section, nous allons présenter notre approche permettant le multiplexage de plusieurs programmes *BPF* indépendamment du point d'accrochage. Notre approche permet de garantir l'ordre d'exécution des programmes. De plus, contrairement à d'autres approches utilisées ou mentionnées dans la littérature. Finalement, notre approche diffère des techniques existantes, car les programmes développés n'ont pas besoin d'avoir connaissance de notre approche pour bien fonctionner.

Cette section présente la chaîne de programmes *BPF* et les opérations qu'il est possible de réaliser dessus, ainsi que les modifications que notre chaîne apporte aux instructions *BPF* des programmes.

## 3.1 Description

La chaîne de programmes *BPF* est une approche pour permettre le séquençage automatique de programmes *BPF* indépendants par l'application d'un correctif aux instructions des programmes compilés permettant le chaînage d'au maximum 33 programmes par appels de queue.

Nous représentons la chaîne de programmes *BPF* par un graphe orienté acyclique formant une liste chaîne de programmes *BPF* à être exécutés par le noyau de *Linux*. Nous identifions ce graphe orienté acyclique en utilisant la terminologie standard, soit $G = (V, E)$, où $G$ représente la chaîne, $V$ les programmes *BPF* de la chaîne, et $E$ les liens entre les programmes.

Dans cette section, nous présentons toutes les opérations réalisables sur la chaîne de programmes *BPF* présentée ci-dessus, soit l'initialisation, l'ajout, la suppression, et le remplacement de programmes dans la chaîne de programmes *BPF*. Pour chaque opération, nous présentons son algorithme et une description de cet algorithme.

Le tableau suivant présente les opérations utilisées par les algorithmes :

Tableau 3.1 Opération commune entre les algorithmes

| Opération | Description |
|---|---|
| *patch* | Applique un correctif au programme *BPF* afin de le rendre compatible avec la chaîne de programmes *BPF*. Ce correctif est documenté dans le sous-chapitre 3.2. |
| *insert* | Insère un programme dans l'ensemble des programmes $V$ composant la chaîne, ou dans l'ensemble des liens entre les programmes $E$. Cette opération inclue le chargement du programme *BPF* dans le noyau de *Linux*. |
| *replace* | Remplace un programme dans l'ensemble des programmes $V$ composant la chaîne, ou dans l'ensemble des liens entre les programmes $E$. Cette opération inclue le chargement du programme *BPF* de remplacement dans le noyau de *Linux*. |
| *remove* | Supprime un programme dans l'ensemble des programmes $V$ composant la chaîne, ou dans l'ensemble des liens entre les programmes $E$. |
| *attach* | Attache le programme au point d'accrochage. Cette opération est implémentée par le noyau de *Linux*. |
| *detach* | Détacche le programme au point d'accrochage. Cette opération est implémentée par le noyau de *Linux*. |

### 3.1.1 Initialisation d'une chaîne de programmes *BPF*

Afin d'initialiser la chaîne de programmes, un ensemble ordonné de programmes $P$ est requis afin de les attacher à un point d'accrochage. L'algorithme 3.1 présente le pseudocode permettant la création d'une chaîne de programmes *BPF*.

Algorithme 3.1 Algorithme d'initialisation d'une chaîne de programmes *BPF*

**Algorithme :** chain_init

**Input :** Ordered set of programs $P \neq \emptyset$, an empty chain $G = (V, E)$, an attach point $A$

**Output :** Updated chain $G = (V, E)$

**for** *all programs* $p_i \in P$ $(i = 0, 1, 2, 3, ...)$ **do**
    patch $p_i$
    insert $p_i$ in $V$
    **if** $i \neq 0$ **then**
        insert $(p_{i-1}, p)$ in $E$
    **end if**
**end for**
attach $p_0$ on $A$

### 3.1.2 Remplacement atomique d'un programme *BPF*

Afin de remplacer un programme de la chaîne de programmes *BPF*, un programme *BPF* de remplacement $p$ ainsi que l'indice $i$ du programme à remplacer sont nécessaires. L'algorithme 3.2 présente le pseudocode de cette opération. L'ordre des opérations dans cet algorithme est primordial afin de garantir que le remplacement est atomique du point de vu des évènements rentrant dans la chaîne de programmes.

Algorithme 3.2 Algorithme de modification d'une chaîne de programmes *BPF*

**Algorithme :** chain_replace

**Input :** An index $i \neq 0$, a program $p' \notin V$, an existing chain $G = (V, E)$, and its attach point $A$

**Output :** Updated chain $G = (V, E)$

$p \leftarrow V_i$
patch $p'$
**if** *exists*$(V_{i+1})$ **then**
    insert $(p', V_{i+1})$ in $E$
**end if**
replace $(V_{i-1}, p)$ with $(V_{i-1}, p')$ in $E$
**if** *exists*$(V_{i+1})$ **then**
    remove $(p, V_{i+1})$ in $E$
**end if**
replace $p$ with $p'$ in $V$

L'algorithme 3.2 ci-dessus présente une méthode de remplacement atomique pour les programmes *BPF* de la chaîne. L'ordre des opérations est primordial afin de garantir l'atomicité de la chaîne de programmes.

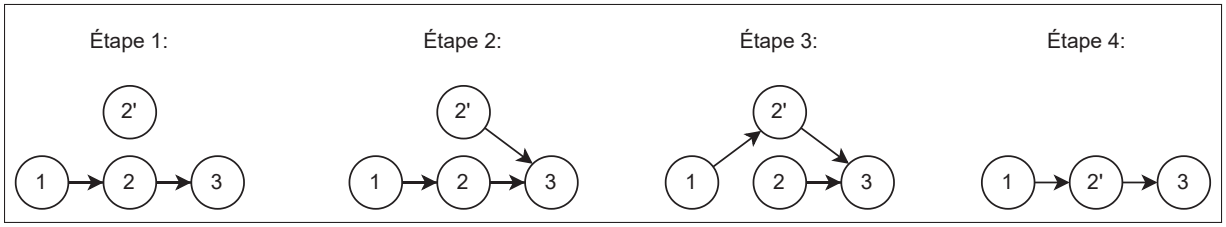


Figure 3.1 Étapes de remplacement de programmes

Comme le montre la figure 3.1, un exemple de remplacement de programme au milieu d'une chaîne de programmes, il est possible d'apercevoir que les programmes consistant en des carrés forment toujours une chaîne ininterrompue de programmes.

### 3.1.3 Recréation d'une chaîne de programmes *BPF*

La section 3.1.2 indique comment remplacer un programme de la chaîne de manière atomique, cependant cette approche présente des lacunes lorsque vient le temps de modifier, ajouter, ou supprimer plus d'un programme à une chaîne de programme.

Dans le cas où il est nécessaire de faire plusieurs opérations modifiant la chaîne de programmes, il est mieux de recréer la chaîne de programmes *BPF* depuis du début afin de ne pas introduire d'instabilité dans la chaîne. Cela peut arriver si au moins 2 programmes sont dépendants les uns des autres.

Algorithme 3.3 Algorithme de recréation d'une chaîne de programmes *BPF*

**Algorithme :** chain_recreate

**Input :** An old chain $G = (V, E)$, an ordered set of new programs $P'$, and an attach point $A$

**Output :** Updated chain $G' = (V', E')$

chain_init $G'$ with $P'$
**if** *A can be atomically replaced* **then**
    replace $G$ with $G'$ on $A$
**else**
    detach $G$ on $A$
    attach $G$ on $A$
**end if**

## 3.2 Correctif appliqué aux programmes *BPF* de la chaîne

Les programmes faisant partie de la chaîne de programmes *BPF* n'ont pas connaissance les uns des autres préalablement à être inséré dans la chaîne. Afin de rendre les programmes compatibles avec la chaîne de programmes *BPF*, il faut appliquer un correctif à tous les programmes de la chaîne avant de les charger dans le noyau de *Linux*. Ce correctif permet à plusieurs programmes *BPF* d'être chaînés sans que les programmes soient individuellement au courant de la présence d'autres programmes dans la chaîne de programmes *BPF*.

Algorithme 3.4 Correctif appliqué aux programmes *BPF* de la chaîne de programmes *BPF*

```
1  Algorithme : patch
   Input : The index i of the program, an existing chain G = (V, E), the contextual
           information of the BPF call ctx, the success codes S, the redirect codes R , and
           a saved result code C
   Output : A BPF result code
2  c ← V_i(ctx)
3  if c ∉ S & c ∉ R then
4  |   return c
5  end if
6  if exists V_{i+1} then
7  |   if c ∈ R then
8  |   |   C ← c
9  |   end if
10 |   return V_{i+1}(ctx)
11 end if
12 if C then
13 |   return C
14 else
15 |   return c
16 end if
```

Le correctif est appliqué en préfixant le programme *BPF* d'instructions afin d'obtenir un comportement similaire à celui présenté par l'algorithme 3.4. Pour pouvoir appliquer le correctif, les informations suivantes sont nécessaires :

1. Une *map* contenant l'information des appels de queue de la chaîne de programmes *BPF*. Cette *map* encode le $E$ de la chaîne de programmes *BPF* ;

2. Une *map* persistant de l'information au travers de la chaîne de programmes. Cette *map* est de type *BPF_MAP_TYPE_PER_CPU_ARRAY* ;
3. L'indice du programme *BPF* dans la chaîne de programmes *BPF* ;
4. Les codes de succès du programme *BPF* ;
5. Les codes de redirection du programme *BPF* ;
6. Les instructions du programme à corriger.

## 3.3 Limitations de la solution

Dans cette section, nous présentons les limitations rencontrées lors de l'implémentation de notre approche, et une présentation de solutions pour mitiger ces limitations.

### 3.3.1 Remplacement du premier programme de la chaîne

L'impossibilité d'appliquer des modifications au premier programme des chaînes de programmes *BPF* est une limitation importante de notre approche. Dût au fonctionnement du noyau de *Linux*, le premier programme de la chaîne doit être attaché à un point d'accrochage dans le noyau de *Linux* afin que la chaîne de programmes *BPF* soit exécutée.

Cela cause un problème lorsqu'il vient le temps de modifier le premier élément de la chaîne de programmes, car il devient impossible de modifier le programme sans détacher la chaîne de programmes *BPF*. Pour cette raison, toute modification au premier programme de la chaîne de programmes *BPF* nécessite la recréation d'une nouvelle chaîne de programmes *BPF* pour remplacer la chaîne de programmes *BPF* existante.

### 3.3.2 Recréation de la chaîne

La nécessité de recréer la chaîne de programmes lors de toute modification plus complexe que le remplacement d'un seul programme est connue. Cette limitation est acceptée, car tentez d'y

pallier entraînera des pertes de performances par l'utilisation d'indirections, et donc des pertes de performances.

Nous assumons que les chaînes de programmes *BPF* sont très souvent lues et rarement modifiées. Ce qui implique que le sacrifice de performances, bien que minimes pour permettre une optimisation des modifications des chaînes de programmes ne fait pas de sens. S'il est cependant désiré d'optimiser la recréation de chaînes de programmes *BPF*, voici deux approches qu'il est possible d'explorer :

Une première solution est la possibilité d'utiliser un niveau d'indirection supplémentaire pour la création d'une *map* de *maps* contenant 1 entrée, la *map* de redirection pour les appels de queue. Recréer la chaîne de programmes va nécessiter de créer une nouvelle *map* de programmes et mettre à jour la seconde *map* permettant l'indirection. Cette solution est la moins performante, mais offre une meilleure compatibilité.

Une seconde solution est l'utilisation d'une sous-fonction *BPF* faisant l'appel de queue. Il est ensuite possible d'utiliser *BPF_F_REPLACE* pour remplacer la sous-fonction *BPF* avec une nouvelle sous-fonction utilisant une *map* différente. Cette solution est la plus performante, mais est limitée ou types de programmes supportant *BPF_F_REPLACE* et l'architecture *x86_64* de processeurs.

# CHAPITRE 4

# MÉTHODOLOGIE D'ÉVALUATION DES APPROCHES

## 4.1 Environnement de test

Notre environnement de test consiste en une machine utilisant un processeur *Intel i9* 10850K et *Arch Linux* comme système d'exploitation. Nous utilisons l'outil *iperf3* afin de générer une charge de travail.

Nous réutilisons aussi les outils de *XDP Project*, un groupe de travail ayant comme mandat l'amélioration des outils *XDP* dans le noyau de *Linux* XDP Project Authors (2018). Nous créons un environnement de test à l'aide des scriptes du didacticiel de *XDP Project*, ces derniers créant une interface réseau virtuelle dans laquelle nous pouvons charger nos programmes *BPF* sans impacter le reste du système.

Afin d'isoler *iperf3* du reste du système, nous isolons 2 cœurs physiques de l'ordonnanceur de *Linux* avec l'option *isolcpus* (ex. : $isolcpus = 0, 1, 2, 3$) lors du lancement du système. Nous allons ensuite configurer le gouverneur de fréquence du processeur en mode *userspace* pour les cœurs logiques isolés ci-dessus en forçant la fréquence des cœurs à 3.9 GHz exactement. Finalement, nous démarrons le client et le serveur *iperf3* en forçant leur affinité de cœurs sur chacun des cœurs physiques isolés (ex. : les cœurs 0 et 2).

Dans le cadre de cette recherche, nous testons 3 méthodes pour charger un trio de programmes *XDP* sur l'interface réseau virtuelle. En utilisant une interface réseau virtuelle, nous pouvons garantir l'exécution des programmes *BPF* par le noyau *Linux* sur les cœurs isolés du reste de notre environnement. Les 3 programmes que nous développons acceptent des paquets *Internet Protocol Version 6* (*IPv6*) combinés avec *Transmission Control Protocol* (*TCP*), et appliquent les fonctionnalités suivantes :

1. Un programme filtrant les paquets *IPv6* et *TCP*
2. Un programme accumulant des métriques sur les paquets ;

3. Un programme *loggant* de l'information sur les paquets.

Pour des raisons de comparaison, notre approche sera comparée avec 2 approches existantes, pour un total de 3 programmes testé par 3 approches différentes, soit :

1. Chaînage manuel ;
2. *XDP Dispatcher* ;
3. Notre approche (Chaîne de programmes *BPF*).

Nous allons ensuite tester ces 3 approches sur toutes les versions *LTS* (*Long Term Support*) du noyau *Linux* comme mentionné sur les archives de *Linux* (Kernel development community (2023c)). Au moment de l'écriture de ce travail, les versions 6.1, 5.15, 5.10, 5.4, et 4.19 de *Linux* sont en *Long Term Support* (*LTS*).

### 4.1.1 Génération de charges de test

La charge de test est générée à l'aide de *iperf*3, un programme pour mesure de la performance de réseaux. Avec *iperf3*, nous tentons de déterminer le débit maximum possible dans la configuration courante. Les résultats obtenus sont la médiane de 50 exécutions.

### 4.1.2 Collection de métriques

Cette section présente la méthodologie de collection des métriques à des fins comparatives lors de l'analyse et de la discussion des résultats.

#### 4.1.2.1 Cycles CPU

Puisque nous voulons comparer la différence en performance de plusieurs approches de chaînage de programmes *BPF*, nous utilisons la métrique la plus représentative de la performance de programmes, soit le nombre de cycles *Central Processing Unit* (*CPU*). De la recherche sur les

coûts des appels en queue dans les programmes *BPF* à la suite des mitigations de *Spectre* dans le noyau *Linux* a été entreprise par Joly & Serman (2020). Nous basons notre approche sur leur deuxième méthode de collecte d'information.

L'approche de Joly & Serman (2020) que nous allons donc suivre est l'utilisation d'un *kprobe* et *kretprobe* sur la fonction *veth_poll* dans le noyau de *Linux*. Nous créons 2 programmes *BPF*, un *kprobe* et un *kretprobe* s'attachant sur la fonction *veth_poll*, et utilisons une *map* de *perf events* comptant le nombre de cycles *CPU*. Ces programmes sont attachés avant de démarrer le programme de test et affichent comme résultat le nombre total de cycles *CPU* utilisé pour l'exécution des programmes *XPD*, ainsi que le nombre de fois que ces programmes se sont exécuté.

#### 4.1.2.2 Temps d'exécution

Joly & Serman (2020) mentionnent qu'utiliser des *maps BPF* et calculer le temps d'exécution en ajoutant cette collecte d'information directement avant et après dans les programmes *BPF* sous test est une approche valide, mais nous rejetons cette méthode, car Joly & Serman mentionne que l'inconvénient de cette approche est qu'elle ne prend pas en compte les appels de queues sans adaptations significatives aux programmes en train d'être testés. Notre approche étant basée sur la modification des instructions de programmes, cette approche rendra beaucoup plus difficile le testage de programmes *BPF* comportant des appels en queue. Joly & Serman présentent aussi une autre approche basée sur le débit maximal de requêtes, mais puisque les programmes *XDP* ne sont pas ordonnancés par *Linux*, cette valeur n'offre pas d'avantages comparativement au temps d'exécution ou au nombre de cycles *CPU*.

Nous allons à la place retravailler la méthode de Joly & Serman aux programmes mentionnés dans la collecte de cycles *CPU*. Nous ajoutons à ces programmes la collecte du temps d'exécution des paquets.

#### 4.1.2.3 Débit de transfert maximum

Le débit maximum de transfert présente une autre vue alternative des performances, cette dernière étant directement liée avec le temps d'exécution et le nombre de cycles *CPU*. Cette métrique est calculée par le programme de test *iperf3* que nous utilisons.

### 4.1.3 Approches comparables testées

Afin de pouvoir mettre en évidence notre approche et comparer ses performances, nous la comparons lors des expériences à deux autres approches comparables.

#### 4.1.3.1 Chaînage manuel

Premièrement, nous utilisons une approche dans laquelle le chaînage fait entre les divers programmes *BPF* est explicite. Nous implémentons nos 3 programmes mentionnés plus haut en appelant manuellement la fonction utilitaire *bpf_tail_call*. Cette approche est testée afin d'avoir une évaluation de base des performances du chaînage de programmes *BPF*.

#### 4.1.3.2 XDP Dispatcher

Notre deuxième approche consiste à utiliser *XDP Dispatcher* afin d'exécuter les 3 programmes séquentiellement. *XDP Dispatcher* est la solution alternative principale dans la littérature à l'approche que nous proposons. Cette approche permet, comme la nôtre, le chaînage de programmes. Nous incluons cette méthode, la principale solution de rechange à notre approche proposée dans la littérature, car elle permet aussi le chaînage de programmes en minimisant leur interconnaissance.

# CHAPITRE 5

# RÉSULTATS ET DISCUSSION

## 5.1 Résultats

Dans cette section, nous vous présentons les résultats des expériences, ainsi qu'une interprétation sommaire de ces résultats avant la discussion du prochain chapitre. Nous commençons par vous présenter les résultats sous forme de tableau, puis nous offrons une interprétation appuyée par des graphiques.

Le tableau 5.1 ci-dessous présente les résultats bruts obtenus avec la méthodologie présentée dans le chapitre précédent.

Tableau 5.1 Tableau des résultats

| Approche | Linux | Paquets totaux | Temps total (ns) | Cycles totaux | Débit (Gb/s) |
|---|---|---|---|---|---|
| Manuelle | 6.1 | 248 810 | 7 345 879 383 | 36 020 691 426 | 7,48 |
| XDP Dispatcher | 6.1 | 239 327 | 7 398 361 032 | 35 753 628 502 | 7,35 |
| Chaîne BPF | 6.1 | 236 805 | 7 366 506 314 | 35 338 584 014 | 7,28 |
| Manuelle | 5.15 | 281 731 | 7 693 613 607 | 37 938 591 292 | 7,57 |
| XDP Dispatcher | 5.15 | 276 960 | 7 727 242 250 | 37 989 658 119 | 7,75 |
| Chaîne BPF | 5.15 | 275 955 | 7 696 729 895 | 37 930 177 211 | 7,81 |
| Manuelle | 5.10 | 369 809 | 7 155 167 823 | 34 930 629 697 | 10,4 |
| XDP Dispatcher | 5.10 | 375 173 | 6 844 771 238 | 33 719 918 824 | 9,89 |
| Chaîne BPF | 5.10 | 388 903 | 6 984 593 234 | 34 309 267 790 | 10,2 |
| Manuelle | 5.4 | 354 769 | 6 880 047 777 | 33 226 656 151 | 9,99 |
| XDP Dispatcher | 5.4 | 381 739 | 7 002 703 555 | 33 887 774 870 | 10,1 |
| Chaîne BPF | 5.4 | 388 668 | 6 984 055 570 | 34 135 392 325 | 10,2 |
| Manuelle | 4.19 | 387 804 | 6 997 367 246 | 34 214 054 811 | 10,2 |
| XDP Dispatcher | 4.19 | 387 205 | 7 003 433 883 | 34 190 418 856 | 10,2 |
| Chaîne BPF | 4.19 | 383 235 | 6 987 809 500 | 33 889 770 425 | 10,1 |

Avec les résultats obtenus, il est possible d'extraire de plus d'information reliée aux performances. Le tableau 5.2 présente les temps et cycles *CPU* moyens par paquets pour toutes les approches et versions de *Linux*.

Tableau 5.2 Tableau des moyennes par paquets

| Approche | Linux | Temps moyen par paquet (ns) | Cycles moyens par paquet |
|---|---|---|---|
| Manuelle | 6.1 | 29524,05 | 144771,88 |
| XDP Dispatcher | 6.1 | 30913,19 | 149392,37 |
| Chaîne BPF | 6.1 | 31107,90 | 149230,73 |
| Manuelle | 5.15 | 27308,37 | 134662,47 |
| XDP Dispatcher | 5.15 | 27900,21 | 137166,59 |
| Chaîne BPF | 5.15 | 27891,25 | 137450,59 |
| Manuelle | 5.10 | 19348,28 | 94455,87 |
| XDP Dispatcher | 5.10 | 18244,31 | 89878,32 |
| Chaîne BPF | 5.10 | 17959,73 | 88220,63 |
| Manuelle | 5.4 | 19393,04 | 93657,16 |
| XDP Dispatcher | 5.4 | 18344,22 | 88772,11 |
| Chaîne BPF | 5.4 | 17969,21 | 87826,61 |
| Manuelle | 4.19 | 18043,57 | 88225,12 |
| XDP Dispatcher | 4.19 | 18087,15 | 88300,56 |
| Chaîne BPF | 4.19 | 18233,75 | 88430,78 |

Les figures 5.1 ci-dessous présentent des sous-graphiques à barres contenant les résultats des tableaux ci-dessus en regroupant les résultats par approche et version de *Linux*.

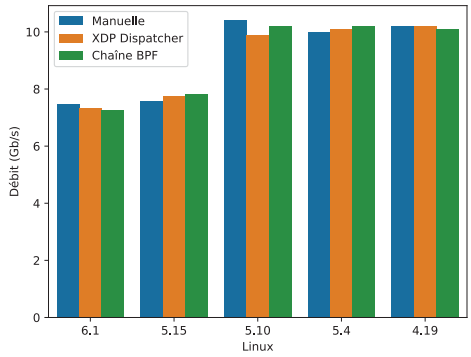


Figure a Débit des approches par version du noyau de *Linux*

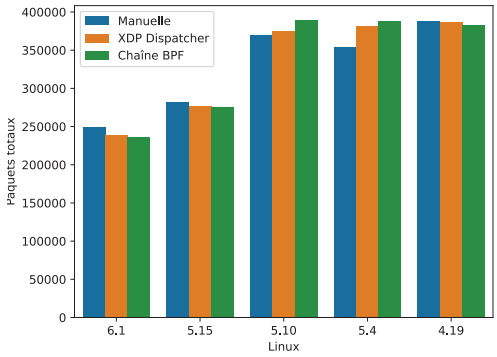


Figure b Nombre de paquets traités des approches par version du noyau de *Linux*

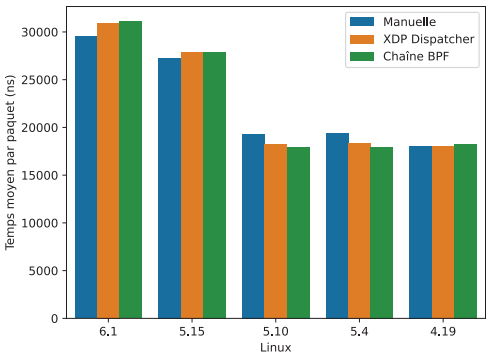


Figure c Temps moyen par paquet des approches par version du noyau de *Linux*

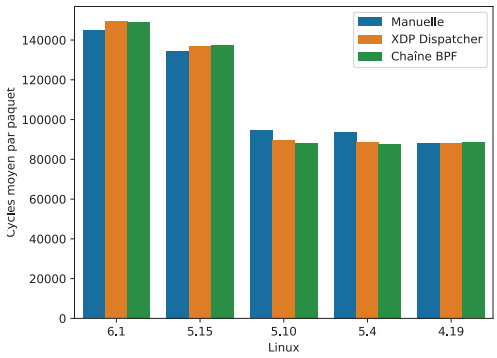


Figure d Cycles moyen par paquet des approches par version du noyau de *Linux*

Figure 5.1 Représentation graphique des tableaux de résultats

Les figures et les tableaux ci-dessus démontrent caractéristiquement que notre approche ne présente pas de différences significatives de performances comparativement aux autres approches actuellement appliquées dans le domaine, soit une différence ayant des impacts négligeables pour l'exécution de chaînes de programmes dans un contexte réel, tels la création de commutateurs *XDP*, la création de *SFC*, etc.

Les résultats obtenus sont donc congruents avec un de nos objectifs, soit de ne pas avoir de différences significatives de performances entre notre approche et les autres approches. Une des

assomptions de cet objectif était que les autres approches présentent les mêmes caractéristiques de performances, ce qui est confirmé par nos résultats.

Il est cependant possible de remarquer une différence significative dans les métriques récoltées après la version 5.4 et du noyau *Linux*. Bien que des améliorations de performances ayant été apportées à la version 5.5 de *Linux*, la version *5.10* voit une perte significative de performance due à un retravail du code permettant les appels de queue tels qu'expliqués par Chaignon (2021).

## 5.2 Comparaison

Cette section présente une comparaison des propriétés des approches testées dans le cadre de ce travail.

### 5.2.1 Compatibilité

Cette section discute de la compatibilité des approches expérimentales testées. Le noyau de *Linux* est rarement directement installé par ses utilisateurs. Les utilisateurs de *Linux* vont souvent faire l'utilisation d'une distribatioution de *Linux*, soit un ensemble de composantes logicielles formant un système d'exploitation complet. Dans bien des cas, ces distributions ne donnent pas le choix de la version et des options de compilation du noyau de *Linux*. Au minimum toutes les approches testées nécessitent un noyau de *Linux* compilé avec du support pour l'exécution de *BPF*.

L'approche par chaînage manuel ne présente aucun problème de compatibilité, peu importe la version du noyau de *Linux* utilisé, ou de l'architecture du *CPU* utilisée.

L'approche utilisant *XDP Dispatcher* présente beaucoup de limitations d'utilisation. Cette approche utilise une fonctionnalité du noyau de *Linux* 5.6 intitulé *BPF Trampoline*. La fonctionnalité *BPF Trampoline* est seulement disponible pour les architectures *CPU* basées sur le jeu d'instruction *x86_64*. La limitation de *XDP Dispatcher* à n'être utilisable que sur les architectures *x86_64* présente un grand défi quant à sont utilisation dans le domaine de la

réseautique et des systèmes embarqués, car ces derniers penchent plus souvent vers l'utilisation d'architectures *CPU* plus écoénergétiques telles *RISC*.

Notre approche présente aussi une limitation sur la version nécessaire du noyau de *Linux* pour son utilisation, soit la version 4.12 de *Linux*, due à l'introduction d'une fonctionnalité permettant l'appel entre différentes fonctions *BPF* par Starovoitov (2017). Comparativement à *XDP Dispatcher*, notre approche présente beaucoup moins de lacunes et est compatible avec un beaucoup plus grand nombre de systèmes. Notre dépendance sur la version 4.12 du noyau de *Linux* n'est pas une limitation importante, car la plus veille version du noyau de *Linux* encore en *LTS* à l'écriture de ce mémoire est la version 4.14, qui est déjà compatible avec notre approche Kernel development community (2023c).

### 5.2.2 Nombre de programmes

Toutes les approches testées dans le cadre de ce travail présentent des limites. Cette section discute donc de ces limites et des contraintes qu'elles peuvent apporter lors de leur utilisation.

L'approche par chaînage manuel et notre approche présentent toutes les 2 la même limite, soit un maximum de 33 appels de queue Kernel development community (2023c). Cette limite est donc le nombre maximum de programmes qu'il est possible de chaîner dans le cadre de ces 2 approches.

*XDP Dispatcher* quant à lui permet un nombre fixe et maximal de 10 programmes. Cette valeur peut cependant être configurée par des options de compilation XDP Project Authors (2018).

Il est donc possible d'argumenter que notre approche et l'approche manuelle sont supérieures, car elles supportent un plus grand nombre de programmes à multiplexer.

### 5.2.3 Points d'accrochages

Toutes les approches ont été testées dans le contexte de programmes *BPF* de type *XPD*. Ce choix n'est pas décisionnel, mais dû à une limitation de *XDP Dispatcher*. Comme son nom l'indique, *XDP Dispatcher* n'est utilisable qu'avec les programmes *BPF* de type *XDP*.

Notre approche et l'approche par chaînage manuel, contrairement, n'ont pas cette limitation, et peuvent être utilisées sur n'importe quel type de point d'accrochages. Notre approche est capable d'attacher une chaîne de programmes *BPF* sur d'autres points d'accrochages tels des l'entrée et la sortie des paquets réseaux dans les *socket*.

Notre approche et l'approche manuelle démontrent clairement une supériorité au nombre de points d'accrochages supportés. Notre approche et l'approche manuelle supportent tous les points d'accrochages que *Linux* offrent, tandis que *XDP Dispatcher* n'offre que les programmes *XDP*.

### 5.2.4 Performance

Comme mentionné dans la section 5.1 sur les résultats des expériences, toutes les approches voient une perte significative de performances entre la version 5.4 et 5.15 du noyau de *Linux*. Malgré cette perte de performance, les performances relatives des 3 méthodes sont très similaires, et ne présentent pas de différences significatives. Il est donc possible de conclure qu'au niveau des performances, aucune approche n'est supérieure aux autres.

## 5.3 Améliorations

Cette section présente des améliorations possibles à la chaîne de programmes. Ces améliorations documentent des solutions à certains problèmes présents actuellement dans l'implémentation expérimentale.

### 5.3.1 Support pour plus de versions du noyau de *Linux*

Comme mentionné dans la section **Compatibilité**, notre approche n'est compatible qu'avec la version 4.12 du noyau de *Linux*. Cette compatibilité est due à l'utilisation d'une fonctionnalité intégrée dans cette version du noyau intitulée «*bpf2bpf function calls*». Cette fonctionnalité permet de faire un appel de fonction vers une adresse du même programme *BPF*.

Nous nous servons de cette fonctionnalité pour isoler l'exécution du programme auquel nous appliquons le correctif. Pour effectuer un appel de queue avec *BPF*, il faut repasser le contexte d'appel, soit la valeur dans le registre $r1$ au début de l'exécution du programme. Il n'y a cependant pas de garantie que le contexte restera dans le registre $r1$ tout au long de la fonction, car le programme *BPF* peut décider d'utiliser ce registre pour de l'arithmétique, un appel de fonction, etc. Dans notre correctif, nous enregistrons le contexte dans le registre $r6$ depuis le registre $r1$, car le registre $r6$, selon l'*ABI* de *BPF*, doit être préservé par le programme.

Il serait possible de supporter des versions du noyau plus basses que 4.12, mais cela demandera beaucoup plus de travail lors de l'application du correctif, incluant faire un suivit des variables et des sorties du programme pour s'assurer que le contexte reste toujours disponible, et que toutes les sorties sont converties.

### 5.3.2 Permettre le remplacement du premier programme

Présentement, notre chaîne de programmes *BPF* a une approche très limitative qui est l'impossibilité de modifier le premier programme de la chaîne sans avoir à reconstruire la chaîne en entier. La reconstruction de la chaîne peut, bien sûr, être optimisée afin de réduire le travail lors de la réinitialisation d'une chaîne, mais une solution plus simple est possible.

Il est possible d'insérer un premier programme dans la chaîne ne faisant qu'un appel de queue vers le premier vrai programme. Cela voudrait dire que le premier programme de la chaîne n'aurait jamais besoin d'être modifié au coût d'une décrémentation du nombre maximum de programmes enchaînable. Le programme ressemblerait à celui présenté en annexe I

## 5.4 Applications

Notre approche, Chaîne de programmes *BPF*, ouvre la porte à plusieurs opportunités d'amélioration de l'écosystème *BPF*. D'autres approches discutées durant la revue de littérature par Tu *et al.* (2017) présentent aussi les mêmes opportunités, mais sont limitées en portabilité par la génération de code *BPF* à partir d'un langage de programmation haut-niveau.

Tu *et al.* (2017) offre un cadriciel incluant un générateur d'instructions *BPF* à partir de code source *P4*, et ils utilisent ce générateur de code pour manuellement chaîner plusieurs programmes *P4* ensemble. Notre approche offre un bon point d'extension pour leur travail, car notre travail leur permettrait de découpler leur générateur de code et du chaînage des programmes. En découplant ces deux éléments, *P4* ne deviennent qu'une façon d'ingérer des programmes, et ils pourraient ainsi supporter bien plus de langages de programmation tels, *C*, *Rust*, *Zig*, etc. Il serait aussi possible d'étendre la solution développée par Tu *et al.* pour permettre l'ingestion de programmes sur d'autres types de programmes que *XDP*. Bien que *XDP* soit un cadriciel puissant pour le domaine de la télécommunication, notre approche offre l'opportunité de développer une application permettant la gestion et le multiplexage de code *BPF* sur tous les types de programmes que supporte le noyau de *Linux*.

Il est donc de notre opinion que les Chaînes de programmes *BPF* rendent possibles plusieurs nouvelles contributions au domaine. Nous surmettons que notre contribution ouvre la porte à un écosystème plus vif et divers. Notre approche rendant possible la composition de programmes indépendants, il devient viable d'implémenter des registres de programmes précompilés, tels les registres d'images de conteneurs, et d'utiliser notre approche pour composer ces programmes.

# CONCLUSION ET RECOMMANDATIONS

Dans ce travail, nous vous avons présenté une nouvelle approche pour automatiquement enchaîner des programmes *BPF* dans le but de séquencer leur exécution sur un même point d'accrochage dans le noyau *Linux*.

Après avoir présenté la méthodologie de notre approche, nous avons démontré par expériences que notre approche présente les mêmes caractéristiques de performance des approches existantes, soit le chaînage manuel et *XDP Dispatcher*. Nous avons aussi discuté des points plus subjectifs de notre approche pour argumenter qu'elle présente de grands avantages subjectifs comparativement aux autres.

Notre travail offre une solution alternative et subjectivement meilleure, car elle permet une meilleure indépendance entre les programmes *BPF* et une meilleure compatibilité pour différentes versions de *Linux* et architectures *CPU*.

Nous espérons que ce travail sera utilisé comme point d'entrée pour améliorer l'écosystème *BPF*. Un chaînage facile de programme *BPF* permettrait une nouvelle façon de consommer des programmes *BPF*. En permettant de déclarer une chaîne composé de programmes orthogonaux, certains environnements tels *XDP* et les programmes pour *socket*s deviennent un bac à sable pour composer divers programmes *BPF* pour répondre à un besoin précis.

## ANNEXE I

## PREMIER PROGRAMME *BPF* ALTERNATIF

```
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, int);
    __type(value, int);
} chain SEC(".maps");

SEC("ext")
int dummy(void* ctx)
{
    bpf_tail_call(ctx, &chain, 0);
    return 0;
}
```

# ANNEXE II

# GRAPHES DES RÉSULTATS

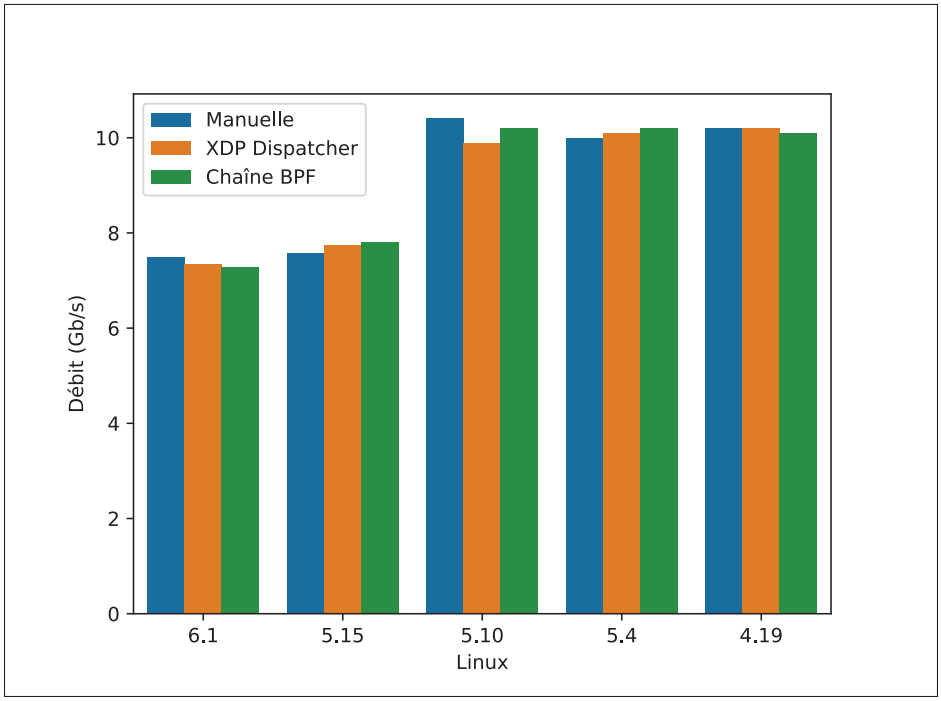


Figure-A II-1 Débit des approches par version du noyau de *Linux*

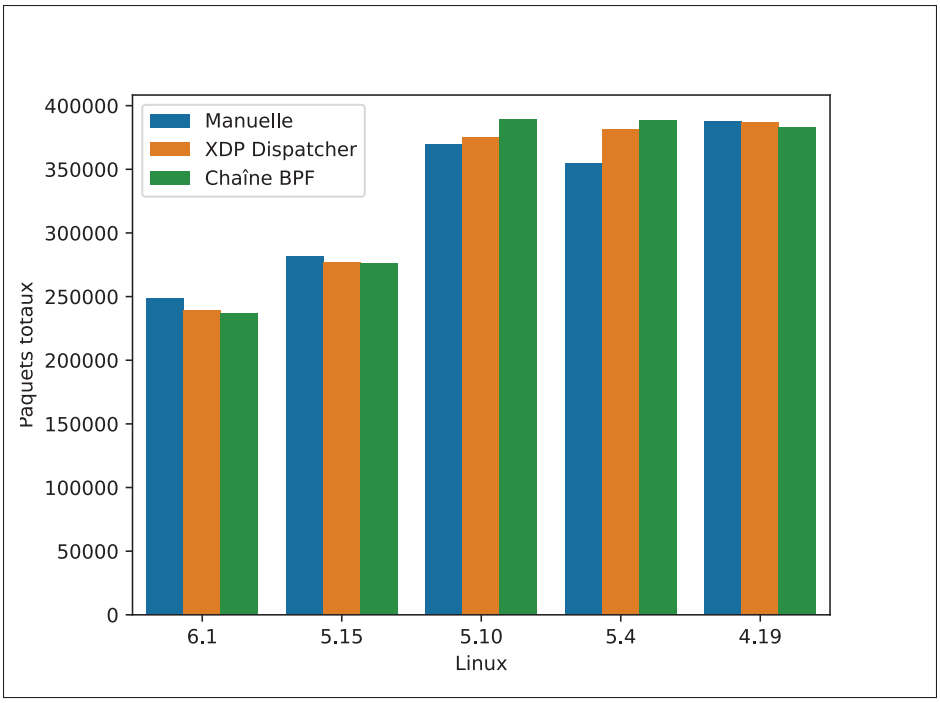


Figure-A II-2 Nombre de paquets traités des approches par version du noyau de *Linux*

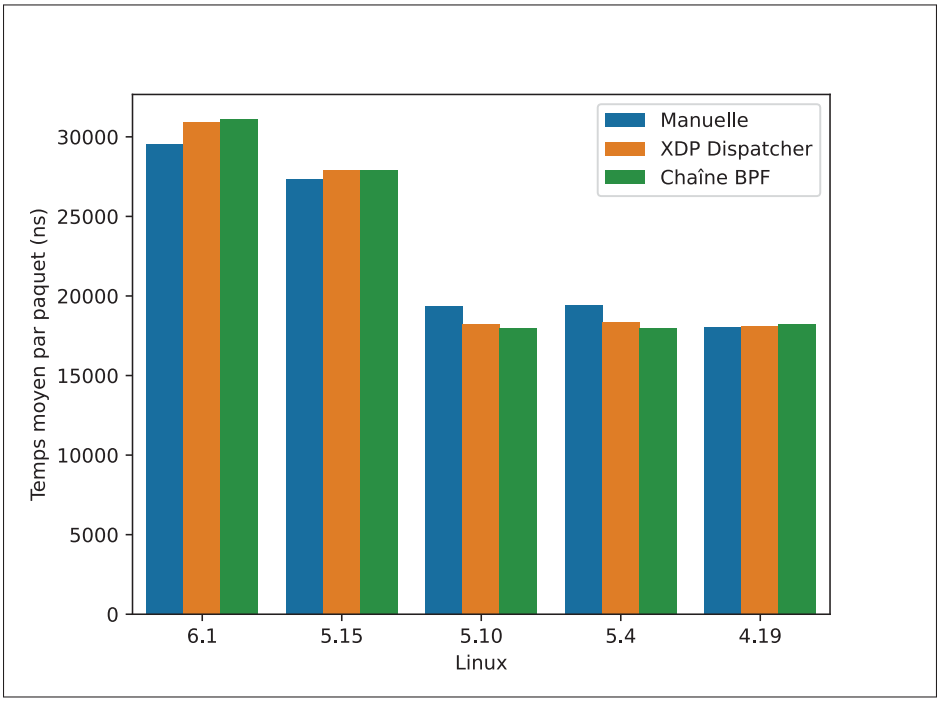


Figure-A II-3 Temps moyen par paquet des approches par version du noyau de *Linux*

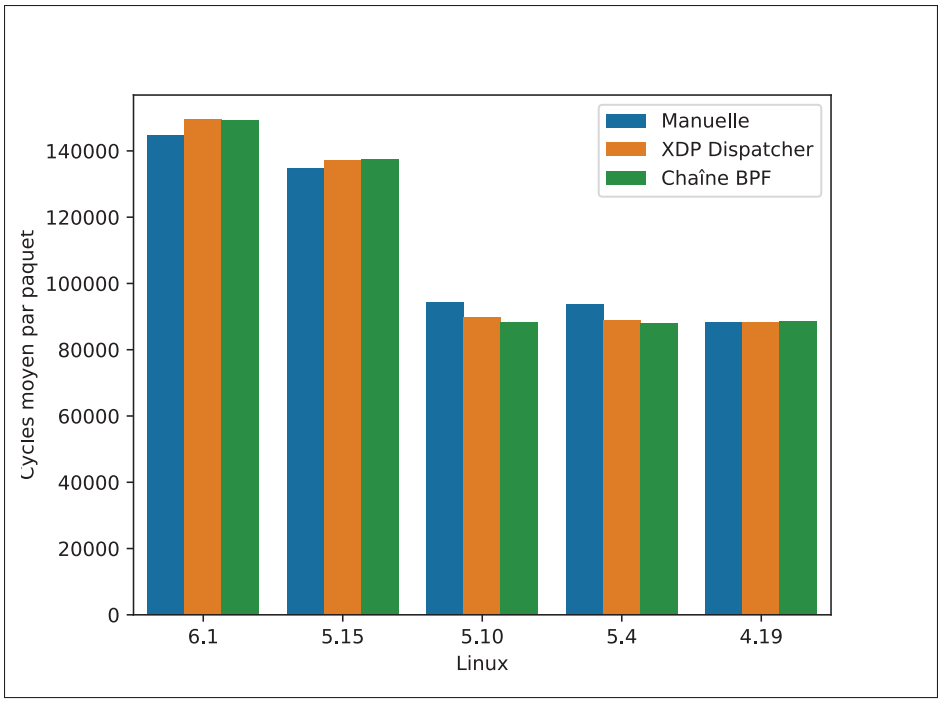


Figure-A II-4 Cycles moyen par paquet des approches par version du noyau de *Linux*

## BIBLIOGRAPHIE


BCC development community. [original-date : 2015-05-01T19 :52 :32Z]. (2023). BPF Compiler Collection. IO Visor Project. Repéré le 2023-06-14 à https://github.com/iovisor/bcc.

Benvenuti, C. (2006). *Understanding Linux Network Internals*. "O'Reilly Media, Inc.".

Bernier, D. (2021). Why eBPF is changing the telco networking space. Bell Canada. Repéré le 2022-05-15 à https://www.youtube.com/watch?v=fNtG0iHYne4.

Bovet, D. P. & Cesati, M. (2002). *Understanding the Linux Kernel*. "O'Reilly Media, Inc.".

bpftool authors. [original-date : 2022-01-16T01 :17 :33Z]. (2023). bpftool. libbpf. Repéré le 2023-06-06 à https://github.com/libbpf/bpftool.

Brodeur, A., Boukhtouta, A., Necir Medakene, A. & Gherbi, A. [P108885]. (2023). BPF Test Framework.

Calavera, D. & Fontana, L. (2019). *Linux observability with BPF : advanced programming for performance analysis and networking* (éd. First edition). Beijing [China] ; Sebastopol, CA : O'Reilly Media, Inc.

Castanho, M. S., Dominicini, C. K., Martinello, M. & Vieira, M. A. M. (2022). Chaining-Box : A Transparent Service Function Chaining Architecture Leveraging BPF. *IEEE Transactions on Network and Service Management*, 19(1), 497–509. doi : 10.1109/TNSM.2021.3122135.

Chaignon, P. (2021). The Cost of BPF Tail Calls. Repéré le 2023-06-19 à https://pchaigno.github.io/ebpf/2021/03/22/cost-bpf-tail-calls.html.

Gregg, B. (2020). *BPF Performance Tools*. Addison-Wesley.

Høiland-Jørgensen, T. & Kartikeya Dwivedi, K. (2023). Protocol for atomic loading of multi-prog dispatchers. Repéré le 2023-06-14 à https://github.com/xdp-project/xdp-tools/blob/master/lib/libxdp/protocol.org.

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D. & Miller, D. (2018). The eXpress data path : fast programmable packet processing in the operating system kernel. *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 54–66. doi : 10.1145/3281411.3281443.

Joly, C. & Serman, F. (2020). Evaluation of tail call costs in eBPF.

Karlsson, M. & Topel, B. (2018). The Path to DPDK Speeds for AF XDP.

Kasatkin, D. (2001). *Affix in a Nutshell*. Repéré à https://affix.sourceforge.net/affix-doc/index.html.

Keniston, J., Panchamukhi, P. S. & Hiramatsu, M. (2022). Kernel Probes (Kprobes) — The Linux Kernel documentation. Repéré le 2023-07-29 à https://docs.kernel.org/trace/kprobes.html.

Kernel development community. (2023a). BPF Type Format. Repéré le 2023-06-14 à https://www.kernel.org/doc/html/next/bpf/btf.html.

Kernel development community. (2023b). eBPF verifier. Repéré le 2023-06-14 à https://docs.kernel.org/bpf/verifier.html.

Kernel development community. (2023c). The Linux Kernel Archives. Repéré le 2023-06-08 à https://www.kernel.org/.

libbpf Authors. [original-date : 2018-10-10T00 :24 :34Z]. (2023). libbpf. libbpf. Repéré le 2023-06-09 à https://github.com/libbpf/libbpf.

Messier, R. (2015). *Operating System Forensics*. Syngress.

Miano, S., Risso, F., Bernal, M. V., Bertrone, M. & Lu, Y. (2021). A Framework for eBPF-Based Network Functions in an Era of Microservices. *IEEE Transactions on Network and Service Management*, 18(1), 133–151. doi : 10.1109/TNSM.2021.3055676.

Mogul, J., Rachid, R. & Accetta, M. (1987). The Packet Filter : An Efficient Mechanism for User-Level Network Code. Repéré à http://www.bitsavers.org/pdf/dec/vax/ultrix-32/4.0_Jun90/AA-PBM2A-TE_Ultrix_4.0_The_Packet_Filter_-_An_Efficient_Mechanism_for_User-Level_Network_Code_Jun1990.pdf.

Rakuten. (2023). eBPF for Telco | Telco Cloud | Observability | Networking | Security. Repéré le 2023-07-10 à https://symphony.rakuten.com/blog/introducing-rakuten-telecom-ebpf.

Raymond, E. S. (2003). *The Art of UNIX Programming*. Addison-Wesley Professional.

Soldani, D., Nahi, P., Bour, H., Jafarizadeh, S., Soliman, M. F., Di Giovanna, L., Monaco, F., Ognibene, G. & Risso, F. (2023). eBPF : A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond). *IEEE Access*, 11, 57174–57202. doi : 10.1109/ACCESS.2023.3281480.

Starovoitov, A. (2017). bpf : introduce function calls (function boundaries) · torvalds/-linux@cc8b0b9. Repéré le 2023-06-08 à https://github.com/torvalds/linux/commit/cc8b0b92a1699bc32f7fec71daa2bfc90de43a4d.

Starovoitov, A. (2022). The untold story of BPF | Kernel Recipes 2022. Repéré le 2023-05-24 à https://kernel-recipes.org/en/2022/talks/the-untold-story-of-bpf/.

Sun Microsystems Inc. (1990). *SunOS 4.1.1 Reference Manual*. Mountain View, CA. Repéré à http://www.bitsavers.org/pdf/sun/sunos/4.1/800-3802-10A_SunOS_4.1_Release_Manual_199003.pdf.

Tu, C.-C., Stringer, J. & Pettit, J. (2017). Building an Extensible Open vSwitch Datapath. *ACM SIGOPS Operating Systems Review*, 51(1), 72–77. doi : 10.1145/3139645.3139657.

Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Júnior, E. P. M. C. & Vieira, L. F. M. (2020). Fast Packet Processing with eBPF and XDP : Concepts, Code, Challenges, and Applications. *ACM Computing Surveys*, 53(1), 16 :1–16 :36. doi : 10.1145/3371038.

Vittal, S. & Franklin, A. A. (2022). HARNESS : High Availability Supportive Self Reliant Network Slicing in 5G Networks. *IEEE Transactions on Network and Service Management*, 19(3), 1951–1964. doi : 10.1109/TNSM.2022.3157888.

XDP Project Authors. (2018). XDP Project. Repéré le 2023-06-06 à https://github.com/xdp-project.

xdp-tools development community. [original-date : 2019-06-23T18 :58 :34Z]. (2023). xdp-tools - Library and utilities for use with XDP. XDP-project. Repéré le 2023-06-14 à https://github.com/xdp-project/xdp-tools.