Faculté des Sciences et Ingénierie

Projet SAR

Comparaison des performances entre un code Berkeley Packet Filter et un module du noyau Linux

Aymeric Ploton, Baptiste Pires, Julian Barathieu, Leif Henriksen Larez

25 mai 2021

Table des matières

1 Introduction

L'architecture des applications a évolué au fil du temps. L'architecture dite *cloud* est aujourd'hui la norme. Il est donc important d'avoir des serveurs performants pour assurer une bonne expérience aux utilisateurs. Il faut également assurer le bon fonctionnement de ces derniers. C'est pourquoi l'utilisation de programmes `eBPF` a grandement augmenté durant les dernières années. Alliant sécurité et performance, ils répondent à ces nouveaux problèmes. Mais ils ont un coût, c'est ce que nous allons évaluer dans ce manuscrit.

Depuis l'apparition du réseau la question s'est posée de savoir comment filtrer correctement et efficacement les paquets de ce premier. S'en est suivie la création de plusieurs logiciels suite à cette demande. L'un des plus connus, l'outils Berkeley Packet Filter (BPF) qui était initialement une technologie sortie en 1992 qui permettait de filtrer les paquets directement depuis le noyau sans avoir à les transférer vers l'espace utilisateur. Sa portée a été étendue grâce à un succès de plus en plus grandissant dans le milieu de la recherche et de l'industrie. Notamment réputé pour ses performances, sa sécurité et sa facilité de développement.

Dans le cadre de ce projet, nous allons nous intéresser aux différences de performances entre un code binaire BPF et son équivalent en module noyau Linux. En effet chacune de ces méthodes a des avantages et des inconvénients qui seront discutés dans les parties suivantes. Cependant nous nous concentrerons sur l'établissement de métrique(s) concernant les différences de performance entre ces deux mécanismes.

Ce projet a été motivé par le fait qu'il n'existe à ce jour pas de métrique ni de comparaison entre ces deux méthodes, nous avons pour but d'en établir des métriques fiables. Pour ce faire nous avons développé une plateforme de test qui compare automatiquement deux codes similaires, l'un est un module Linux, l'autre un progamme `eBPF`. Grâce à cette plateforme nous avons pu mener une campagne d'évaluation basée sur des *micro-benchmarks* conçus pour comparer la performance des deux méthodes.

Dans la suite de ce manuscrit nous allons en premier lieu parler des modules Linux, que nous allons utiliser comme *baseline* pour comparer les programmes `eBPF`. En deuxième lieu nous allons décrire d'une façon détaillée `eBPF`, son fonctionnement et comment l'utiliser. Ensuite en troisième lieu nous allons présenter notre plateforme de test, utilisée pour comparer les modules Linux et les programmes `eBPF`. Puis en quatrième lieu nous allons étudier les résultats obtenus après l'exécution de nos test. Et pour finir, en cinquième lieu nous allons faire une conclusion sur les différentes parties du manuscrit.

2 Modules Linux

Le noyau Linux étant du type monolithique modulaire, il permet l'insertion de programmes dans le noyau durant son exécution. Ce sont des modules qui sont insérés. Ces derniers sont des fichiers binaires compilés. Il en existe deux types, le premier, dit *statique*, comprend les modules compilés avec le noyau et n'ont pas à être insérés pendant son exécution. Le second, les modules *dynamiques*, sont insérés durant son exécution. Ces derniers peuvent également être déchargés dynamiquement.

2.1 Fonctionnement

Les modules une fois chargés (statiquement ou dynamiquement) sont donc présents directement dans l'espace mémoire du noyau. Ils ont donc des droits privilégiés contrairement aux programmes s'exécutant dans l'espace utilisateur. De plus ils ne sont pas vérifiés avant d'être insérés dans le noyau. C'est pour cela qu'il faut être vigilant lorsqu'on développe des modules car ils peuvent présenter de nouvelles failles de sécurité et augmenter la surface d'attaque.

2.2 Avantages et inconvénients

Avantages :

Les modules noyau peuvent êtres chargés et déchargés à "chaud" dans le noyau Linux, s'ils ne font pas partie de l'image binaire de celui-ci. Les modules peuvent accéder à des symboles que le noyau a exportés. De ce fait ils peuvent modifier directement certaines données telles que des variables, des pointeurs, etc.

Ils permettent également de créer de nouvelle fonctionnalités pour le noyau, des pilotes de périphériques, voire d'autre modules.

Inconvénients :

Le fait de charger un module peut générer de la fragmentation mémoire, pouvant entraîner des baisses de performances mineures telles qu'une augmentation des entrées dans le *Translation lookaside buffer (TLB)*.

Ils peuvent également être source de failles de sécurité. En effet, aucune vérification n'est effectuée lors de leur insertion. De ce fait, ils peuvent partiellement ou totalement compromettre le noyau. Il n'est donc pas envisageable de laisser cette liberté à un utilisateur lambda.

En outre, le noyau Linux limite l'accès aux symboles exportés par le noyau selon la licence qui s'applique au code. S'il est *GPL* compatible il bénéficie de tous les symboles explicitement exportés avec `EXPORT_SYMBOL_GPL`. Au lieu de ceux exportés par `EXPORT_SYMBOL` qui regroupe des symboles limités permettant des fonctionnalités *ad-hoc*, simples, permettant juste le chargement notamment.

Enfin Linux ne fournit aucune garantie sur la stabilité de l'`API` ou de l'`ABI` de l'espace noyau. Il peut y avoir des différences structurelles ou fonctionnelles. Il faut vérifier la version des modules à l'aide du progamme `modinfo`.

3 BPF (Berkeley Packet Filter)

Une alternative aux modules est l'insertion de code `eBPF`. A l'origine créé pour l'analyse de paquet réseau, maintenant son usage s'est diversifié. Il apparaît donc comme une alternative à l'utilisation des modules. Dans cette section nous allons présenter son historique, puis son fonctionnement et enfin ses limites.

3.1 Historique

BPF est une machine virtuelle créée par Steve McCanne et Ban Jacobson en 1990 puis publiée pour la première fois en 1992[1]. Le but premier était de fournir une interface bas niveau pour la capture de paquets réseau plus performante que les différentes technologies déjà existantes d'où sa présence dans la plupart des systèmes UNIX. Elle fut intégrée au noyau Linux en 1997 dans la version 2.1.75 par Jay Schlist dans une forme minimale avec seulement 2 registres 32 bits dont un accumulateur et un registre d'index, ainsi qu'une pile de 512 bits appelée "scratch memory". Pour des raisons de sécurité, elle ne supportait pas les instructions de sauts en arrière, ce qui permettait de garantir que tous les programmes se terminent toujours, sachant que la taille d'un programme était limitée.

En 2013, Alexei Starovoito propose Extended BPF [2] (implanté dans le noyau entre la version 4.15 et 4.8), fournissant en plus les fonctionnalités suivantes :

- Une étape de vérification interdisant les boucles infinies, l'accès à des pointeurs non vérifiés, et autres opérations non-sécurisées.
- Un compilateur Just In Time permettant de transformer du *bytecode* BPF en un code machine natif (à l'origine intégré dans le noyau 3.0 en 2011).
- Une amélioration de la machine virtuelle avec le passage de 2 registres 32-bits à 10 registres de 64-bits, et l'intégration d'appels de fonctions noyaux limités.

Il s'ensuivit d'autres améliorations telles que la création d'espace de mappage pour notamment l'échange de données entre le logiciel BPF et le programme utilisateur l'appelant. Ce qui explique son utilisation dans des utilitaires historiques tel que tcpdump.

3.2 Fonctionnement

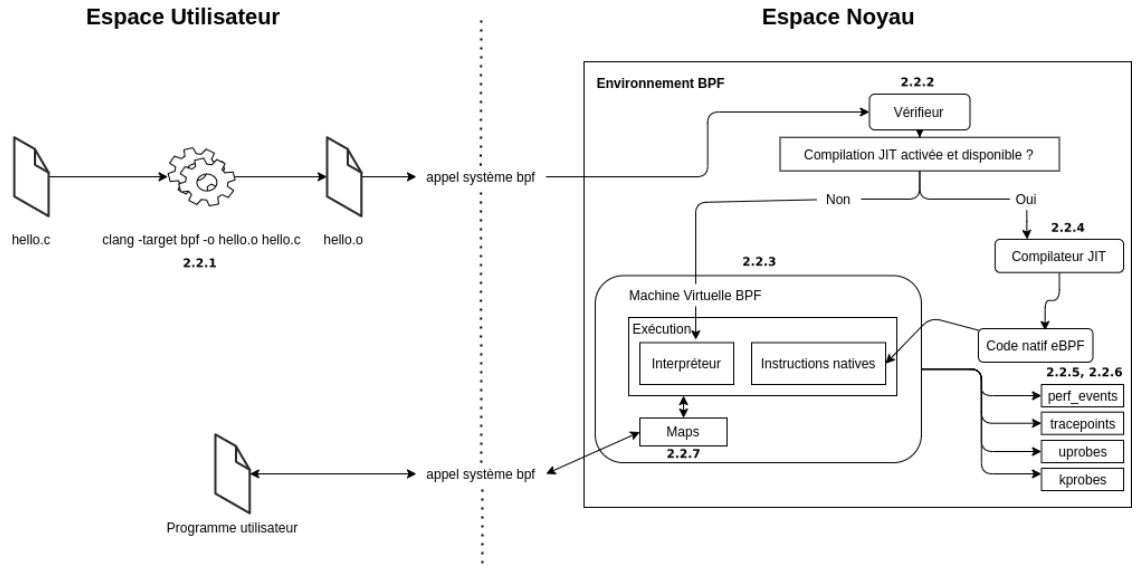

FIGURE 1 – Chaîne de compilation, vérification et d'exécution d'un programme eBPF nommé hello.c (basée sur le livre "BPF Performance Tools", p.19 [3])

Nous allons voir dans cette partie la chaîne de compilation d'un programme eBPF. De sa compilation en passant par la vérification ainsi que le JIT jusqu'à son exécution. Le contenu du fichier hello.c dans la Figure 1 est celui ci-dessous :

hello.c

```c
#include <linux/bpf.h>
#define SEC(NAME) __attribute__((section(NAME), used))

SEC("tracepoint/syscalls/sys_enter_execve")

int bpf_prog(void *ctx) {
    char msg[] = "Hello, BPF World!";
    bpf_trace_printk(msg, sizeof(msg));
    return 0;
}
char _license[] SEC("license") = "GPL";
```

load.c

```c
#include <stdio.h>
#include <uapi/linux/bpf.h>
#include "bpf_load.h"

int main(int argc,char **argv) {
    if(load_bpf_file("hello_world_kern.o") != 0) {
       printf("The kernel didn't load the BPF program\n");
       return-1;
    }
    read_trace_pipe();
    return 0;
}
```

Dans la suite de ce document nous allons voir tout le processus de la compilation à l'injection du *bytecode* eBPF dans le noyau en détaillant les mécanismes internes à eBPF.

3.2.1 Compilation et injection dans le noyau

Les programmes eBPF sont initialement écrits en langage **C**, mais seulement un sous-ensemble de celui-ci est disponible pour des raisons de sûreté et de sécurité. Ces deux aspects seront détaillés dans les sections suivantes.
Pour compiler un code C en *bytecode* eBPF, plusieurs outils sont disponibles :

— Le compilateur Clang
— bcc
— bpftrace

Dans ce document nous allons utiliser **Clang**. Cet outil est le plus simple, vu qu'il ne définit pas de sur-couche d'interface pour injecter les programmes dans le noyau, cela sera notre responsabilité. **bcc** quant à lui, permet d'utiliser le langage *Python* pour injecter des codes *C* eBPF dans le noyau, il prend en charge la compilation ainsi que l'injection dans le noyau. **bpftrace** définit un nouveau langage de plus haut niveau que le C. Il prend également en charge la compilation de ce langage en bytecode eBPF ainsi que son injection dans le noyau.

Dans la figure 1, la compilation est représentée par la ligne *clang -target bpf -o hello.o hello.c*. Elle va produire un fichier *hello.o* (qui contient le bytecode eBPF) qui va par la suite être transféré au noyau Linux à l'aide de l'appel système **bpf**. C'est à la suite de cet appel système que le programme va être vérifié ainsi que compilé à l'aide du `JIT` s'il est disponible et activé.

3.2.2 Vérifieur

Un validateur de code appelé *vérifieur* eBPF analyse le *bytecode* lors de l'insertion du programme dans le noyau et rejette toute opération non-sécurisée. Les opérations suivantes sont considérées non-sécurisées :

— Les boucles infinies (les boucles bornées sont autorisées depuis la version du noyau 5.3).
— Les sauts en arrière.
— Les branchements successifs.

Le validateur rejette les programmes de plus de 4096 instructions. Il simule aussi chaque instruction afin de vérifier que l'état de la pile et des registres soit toujours valide.

Finalement, un vérificateur statique vérifie que le programme n'essaye pas d'accéder à des structures de données du noyau ou de provoquer des erreurs de page.

3.2.3 Machine virtuelle

Une fois le code vérifié, il est prêt pour être interprété par la machine virtuelle. Nous allons voir de quoi celle-ci est constituée et son fonctionnement.

Encodage

Le bytecode eBPF est fait d'un jeu d'instructions assez simple : des opérations arithmétiques ; des instructions de branche ; des appels à des fonctionnalités limitées du noyau.

Toutes les instructions eBPF sont encodées sur 64 bits de la manière suivante, du bit de poids faible au bit de poids fort :

- *opcode* : 8 bits d'opcode, indiquant l'instruction.
- *dst* : identifiant du registre destination, 0-10.
- *src* : identifiant du registre source, 0-10.
- *offset* : 16 bits d'offset.
- *immediate* : 32 bits d'immédiat.

Tous les champs sont présents pour toute instruction. Si une instruction n'utilise pas un champ particulier, il devrait être initialisé à zéro. Dans tous les cas donc l'instruction est longue de 64 bits.

Instructions Le jeu d'instruction du bytecode eBPF contient les instructions suivantes :

- Des instructions arithmétiques et logiques, toutes prenant en argument un registre de destination, et soit un registre de destination, soit un immédiat 32 bits. Ces instructions sont `ADD`, `SUB`, `MUL`, `DIV`, `AND`, `OR`, `XOR`, `LSH` (décalage à gauche), `RSH` (décalage à droite ; logique), `ARSH` (décalage à droite ; arithmétique), et `MOV` (déplacement). On a aussi `NEG` (négation ; arithmétique), qui ne prend qu'un argument (un registre).
- Des instructions arithmétiques et logiques 32-bits, qui n'affectent que les 32 bits les plus bas des registres. Leurs noms sont identiques aux instructions 64-bits correspondantes, mais avec "32" rajouté à la fin, e.g. `ADD32`. Les 32 bits de poids fort du registre de destination sont automatiquement mis à zéro par ces instructions.
- Des instructions d'échange de bytes (byteswaps) : `LE16`, `LE32`, `LE64`, et `BE16`, `BE32`, `BE64`.
- Des instructions de saut : `JA` qui saute non-conditionnellement un offset 16-bits plus loin ; `CALL` qui appel une fonction se trouvant un offset plus loin ; des sauts conditionnels de la forme `J<condition> reg1, reg2/imm, offset` où `<condition>` est une chaine parmis `EQ`, `NE`, `GT`, `GE`, `LT`, `LE` (non-signés), `SET` (si reg1 AND reg2/imm), et `SGT`, `SGE`, `SLT`, `SLE` (signés).
- Une instruction `EXIT` pour quitter le programme, ou la fonction (si appelé par CALL), et renvoyer r0.
- Des instructions spéciales pour les maps. Voir section sur les maps. `LDDW`, `LDABSW`, `LDABSH`, `LDABSB`, `LDINDW`, `LDINDH`, `LDINDB`, et `LDDINDW`.

— Des instructions de chargement mémoire : `LDXDW` qui effectue `dst = *(u64 *)(src + off)`, ses équivalents `LDXW`, `LDXH`, `LDXB` qui sont 32-bits, 16-bits et 8-bits respectivement.

— Des instructions d'écriture d'immédiats en mémoire : `STDW` qui effectue `*(u64 *)(dst + off) = imm`, et ses équivalents `STW`, `STH`, `STB` qui sont 32-bits, 16-bits et 8-bits respectivement.

— Des instructions d'écriture de registres en mémoire : `STXDW` qui effectue `*(u64 *)(dst + off) = src`, et ses équivalents `STXW`, `STXH`, `STXB` qui sont 32-bits, 16-bits et 8-bits respectivement.

Registres Un ensemble de 10 registres R0-R9, tous 64-bits, et un registre *read-only* R10 indiquant le début de la pile.

Les 10 registres 64-bits de eBPF sont :

r0	Contient la valeur de retour des appels au noyau ("*helpers*"), et la valeur de retour du programme eBPF. La signification de la valeur de ce registre dépend du type de programme eBPF. Équivalent au registre "RAX" de l'architecture x86-64.
r1-r5	Contiennent les arguments pour les appels de fonction, à la fois les appels au noyau et les appels eBPF-vers-eBPF. A l'initialisation du programme, r1 est initialisé avec le pointeur vers le contexte actuel. Équivalents aux registres "RDI", "RSI", "RDX", "RCX" et "R8" respectivement.
r6-r9	Registres de travail, préservés par les appels au noyau. Équivalents aux registres "RBX", "R13", "R14", et "R15" respectivement.
r10	Pointeur vers la pile de 512 octets du programme eBPF. Équivalent au registre "RBP" de l'architecture x86-64.

3.2.4 Compilateur JIT

Le noyau Linux dispose d'un compilateur à la volée de eBPF appelé le *Just In Time compiler* (*JIT compiler*). Ce compilateur traduit le *bytecode* eBPF en code **natif**, et ce au moment de l'exécution ; une compilation au runtime, donc. Le JIT est capable de réaliser des optimisations optionnelles dépendantes de la machine et permet d'accélérer considérablement l'exécution du programme eBPF.

Cette technologie s'oppose à celle de la compilation anticipée ou "*Ahead Of Time*" (*AOT*), comme le *jaotc* qui traduit le langage en code machine avant l'exécution du programme qui fournit une vue d'ensemble. Les optimisations rendues possibles par la compilation AOT permettent de diminuer le coup des appels de fonction ou méthode, interface, exception, et autres, là où le JIT ne peut faire que des optimisations locales.

Avec le compilateur JIT les équivalences notées entre les registres eBPF et les registres x86-64 sont importantes : pour une compilation JIT sur une machine utilisant l'architecture x86-64. Il y aura une traduction directe des registres eBPF vers les registres x86-64 correspondants. La compilation JIT des instructions se déroule de manière similaire : un bytecode "ADD" de deux registres eBPF se traduira en une instruction x86-64 "ADD" des registres x86-64 correspondants. Il s'agit d'une traduction simple, instruction par instruction.

Le compilateur eBPF JIT est également disponible sur d'autres architectures que x86-64 : en particulier SPARC, PowerPC, ARM, arm64, MIPS, et s390. Comme le JIT utilise du code assembleur, il faut évidemment une implémentation spécifique pour chaque architecture de processeur. Pour les architectures pas encore supportées par le JIT, un interpréteur existe capable d'exécuter

du bytecode eBPF.

Le compilateur eBPF JIT inclut par le noyau Linux est en fait désactivé par défaut pour des raisons de sécurité. En effet, l'attaque Spectre peut exploiter le compilateur JIT pour extraire des données d'autres processus du noyau et permettre à un processus utilisateur de les lire. Il est également possible de provoquer une exception fatale dans le noyau et ainsi mener un déni de service. Pour activer le JIT eBPF, il faut donc le compiler le noyau avec l'option HAVE_BPF_JIT, puis à l'exécution effectuer la commande "echo 1 > /proc/sys/net/core/bpf_jit_enable".

3.3 Utilisation

Dans cette section nous allons voir les différentes fonctionnalités qui sont à notre disposition afin d'écrire des programmes eBPF ainsi que l'injection dans le noyau.

3.3.1 Kernel helpers

Les *kernel helpers* sont les fonctions du noyau accessibles depuis un programme eBPF afin d'interagir avec le système. Par exemple, ils peuvent être utilisés pour imprimer des messages de débogage, pour obtenir le temps écoulé depuis le démarrage du système, pour interagir avec des *maps* eBPF ou pour manipuler des paquets réseau. Pour un code eBPF compilé en JIT, les *helpers* sont déjà compilés et directement appelés par le code machine correspondant, plutôt que de passer par une API du noyau, et ont donc de très bonnes performances. Les *helpers* sont définis dans le noyau même, pas dans des modules, par une API similaire à celle des appels système. Dû aux conventions eBPF, un *helper* ne peut pas avoir plus de cinq arguments.

Un exemple basique et utile de *helper* est `bpf_trace_printk`, qui permet simplement d'afficher une chaîne formatée à l'écran :

```
long bpf_trace_printk(const char *fmt, u32 fmt_size, ...)
```

3.3.2 Évènements

Il est possible d'utiliser de la programmation événementielle dans un programme eBPF, notamment avec les *kprobes*, *uprobes*, et *tracepoints*.

Les *kprobes* permettent de déclencher l'exécution du programme eBPF avec l'appel de presque n'importe quelle routine du noyau et de collecter les informations de débuggage et de performance de manière non-disruptive. Plus précisément, on peut attacher une *kprobe* à une adresse dans le noyau, et à chaque fois que le code à cette adresse est exécuté, le programme eBPF est exécuté.
Un autre type de probe est le *kretprobes*. Alors qu'un *kprobe* peut être inséré sur n'importe quelle instruction du noyau, un *kretprobe* se déclenche plutôt lors du retour de fonction spécifique, même si la fonction a plusieurs instructions de retour.
Les *uprobes* ont le même objectif que les *kprobes* mais pour l'espace utilisateur. Elles permettent donc d'attacher un programme eBPF à une adresse utilisateur donnée.
Les *tracepoints* sont des *breakpoints* compilés dans un programme statiquement, contrairement à une probe qui est dynamique et placée dans du code arbitraire. Un *tracepoint* peut aussi être relié à un programme eBPF.

3.3.3 BPF Maps

L'utilisation des *eBPF maps* permet de sauvegarder un état entre les différentes invocations d'un programme eBPF, de partager de données entre des programmes eBPF, et également de partager des données entre le noyau et l'espace utilisateur.

Ce sont des tableaux associatifs génériques permettant de garder des clefs et valeurs de taille arbitraire.

Ci-dessous nous présentons une version simplifiée de l'API maps. Les signatures des fonctions changent en fonction d'où elles sont utilisées (espace utilisateur, espace noyau, dans la machine virtuelle). Dans les trois cas, l'API définit toujours au moins trois fonctions : la recherche, la mise à jour et la suppression d'un élément [4].

```
bpf_map_lookup_elem(map, clef, valeur);
bpf_map_update_elem(map, clef, valeur);
bpf_map_delete_elem(map, clef, valeur);
```

3.3.4 Injection d'un programme dans le noyau

Les programmes eBPF sont écrits dans l'espace utilisateur et insérés dynamiquement dans le noyau en utilisant l'appel système `bpf` [5].

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Les paramètres de l'appel système `bpf()` sont les suivants : *cmd* indique code de l'action à réaliser, et a comme valeurs possibles les constantes `BPF_...` ; *attr* contient les paramètres de l'action, la structure exacte dépend de l'action à exécuter ; *size* indique la taille de la structure *attr*, en octets.

La commande `BPF_PROG_LOAD` permet de charger un programme eBPF. Pour l'utiliser, la structure à passer en deuxième argument à`bpf` est la suivante :

```
linux/bpf.h:
struct {    /* Used by BPF_PROG_LOAD */
    __u32         prog_type;
    __u32         insn_cnt;
    __aligned_u64 insns;      /* 'const struct bpf_insn *' */
    __aligned_u64 license;    /* 'const char *' */
    __u32         log_level;  /* verbosity level of verifier */
    __u32         log_size;   /* size of user buffer */
    __aligned_u64 log_buf;    /* user supplied 'char *' buffer */
    ...
};
```

Les types de programmes que l'on peut passer avec `BPF_PROG_LOAD`, à mettre dans le champ `prog_type`, sont les suivants :

BPF_PROG_TYPE...	Description
SOCKET_FILTER	Un filtre de paquet réseau.
KPROBE	Contrôle l'activation d'une kprobe.
SCHED_CLS	Un classifieur de contrôle de trafic réseau.
SCHED_ACT	Une action de contrôle de trafic réseau.
TRACEPOINT	Contrôle un tracepoint.
XDP	Un filtre de paquet réseau lancé depuis le driver récepteur.
PERF_EVENT	Controle un perf event handler.
CGROUP_SKB	Un filtre de paquet réseau pour un group de controle.
CGROUP_SOCK	Comme CGROUP_SKB, mais peut modifier les options socket.
LWT_*	Un filtre de paquet réseau pour les tunnels lightweight.
SOCK_OPS	Un programme servant à choisir les options socket.
SK_SKB	Un filtre de paquet réseau pour rediriger les paquets entre sockets.
CGROUP_DEVICE	Détermine si une opération sur un périphérique est permise ou non.

3.4 Intérêt et limites

Originellement, l'objectif premier de (classic) BPF était d'améliorer le filtrage des paquets réseaux, à l'époque de sa sortie la plupart des filtres tournaient en espace utilisateur, ce qui demandait une copie des paquets depuis l'espace noyau vers l'espace utilisateur. Ces copies ne sont plus nécessaires avec BPF : une machine virtuelle permettant d'interpréter du bytecode BPF qui a été ajoutée au noyau, il devient possible de filtrer les paquets directement dans le noyau.[1].

Après son extension en 2013, il est maintenant possible d'utiliser eBPF pour bien plus que le filtrage des paquets réseaux. Par exemple, eBPF est utilisé par de nombreux outils de traçage de performances.

Avantages :

Le code étant vérifié, les risques d'insérer un code qui pourrait compromettre le noyau sont minimes. Cela peut permettre à des utilisateurs n'ayant pas les droits d'un super-utilisateur de pouvoir insérer du code dans le noyau Linux. C'est possible grâce aux *capabilities* qui permettent à un utilisateur d'avoir accès à certains appels système définis préalablement grâce à des sous-classes notamment via *cgroup*, dont eBPF peut faire partie (avec plus ou moins de finesse) sans avoir besoin d'être *root*.

L'utilisation d'une machine virtuelle permet de rendre les programmes portables, sans se soucier de l'architecture sur laquelle s'exécute le noyau. Il suffit de compiler une fois le programme pour l'insérer des les noyaux compatibles. Le support du JIT pour les architectures supportées permet également d'améliorer les performances voire d'égaler celles du noyau.

Les *maps* permettent également une communication simplifiée avec l'espace utilisateur. Leur mise en place est plus simple que celle des autres mécanismes pour les modules Linux (par exemple le *sysfs*). Mais le coût de cette simplicité se retrouve dans la limitation des fonctionnalités des *maps*.

Inconvénients :

Malgré les nombreuses possibilités offertes par cette technologie, eBPF a néanmoins ses inconvénients et limitations.

Par exemple, le vérificateur de programme n'est pas encore à l'abri des faux positifs, et il ne peut pas être utilisé pour des programmes de grande échelle possédant de nombreux embranchements.

Les critères de sécurité font que la taille des programmes ne doit pas dépasser 4096 instructions (motivé par le fait que le programme s'exécute sur un seul fil d'exécution, la taille est donc liée au temps d'exécution) et limitent très fortement les accès à certaines fonctionnalités du noyau. Il est par exemple interdit d'accéder à l'espace utilisateur ainsi qu'à un quelconque service tiers durant l'exécution du programme. L'accès à certaines structures de données, les allocations de mémoire dynamiques et les sauts indirects sont interdits.

Le déploiement de programme eBPF dans des serveurs exige en général qu'il s'exécute dans des NFV (Network Function Virtualization) qui s'exécutent dans des machines virtuelles. Cela accroît la sécurité mais ce gain est répercuté sur les ressources telles le processeur et la mémoire.

3.5 Conclusion

Ainsi, comme nous l'avons vu en étudiant les modules noyau et les programmes `eBPF`, on comprend pourquoi ces derniers sont très utilisés du fait de leur sécurité, implémentables et utilisables. Côté espace utilisateur, ils possèdent une grande probabilité, et permettent de capturer des données au niveau du noyau lui-même. C'est pour ces raisons qu'ils s'imposent dans une utilisation très large comme par exemple :

- Le filtrage réseau (haute-performance (anti-DDOS, pare-feu distribué, ..)
- La détection d'événement du noyau (mesure de performance, trace, ...)
- Des fonctions avancées et dépendantes du contexte (équilibrage de charge, ...)

Pour bénéficier de ces propriétés, ils nécessitent plusieurs mécanismes tels qu'un vérifieur, un compilateur *JIT*, un interpréteur, ...
Nous pouvons donc nous demander le coup de ces mécanismes par rapport à une exécution normale.

4 Plateforme de test

Pour répondre à la problématique de notre projet, nous avons développé notre propre plateforme de test afin de collecter les ressources utilisées et le temps d'exécution des différentes étapes des deux technologies.

4.1 Les mesures des programmes eBPF

Dans un programme `eBPF` nous pouvons distinguer 3 grandes étapes que nous allons mesurer, qui sont :

La compilation Le temps de compilation total sera mesuré, c'est à dire lors de l'appel à la commande `make`. Les fichiers sont compilés à l'aide d'un fichier `Makefile` qui permettra d'inclure toutes les dépendances nécessaires. Les fichiers doivent également être compilés avec l'option `-O2` de `clang`. Les aspects techniques seront développés plus tard.

L'insertion Ensuite, nous mesurerons le temps d'insertion dans le noyau qui est réalisé à l'aide de la fonction `load_bpf_file`. Durant cet `appel système`, plusieurs fonctions sont appelées pour insérer le programme. Deux d'entre elles vont nous intéresser. La première, `bpf_check`, comme son nom l'indique, réalise la vérification du programme. Une fois cette fonction terminée, la fonction `bpf_int_jit_compiile` est appelée. Celle-ci, si le `JIT` est activé, va transformer le `bytecode eBPF` en code natif machine.

Nous allons mesurer les temps d'exécution de ces deux fonctions à l'aide de l'outil de traçage du noyau `ftrace`. Il nous permet de récupérer les durées d'exécution de fonctions du noyau. La granularité temporelle minimale de `ftrace` est la microseconde, nous avons donc choisi cette échelle pour mesurer les différents temps. Si celle-ci s'avère être trop petite, nous utiliserons les millisecondes.

L'exécution Finalement, nous allons mesurer le temps d'exécution du programme. Pour ce faire, nous allons utiliser l'helper `bpf_ktime_get_ns`, qui fait appel à la fonction du noyau `ktime_get_mono_fast_ns`. Cela nous permettra d'avoir une symétrie entre les mesures faites dans les programmes `eBPF` et dans les modules Linux.

Pour mesurer le temps supplémentaire dû au programme, nous allons englober celui-ci entre deux appels à cet *helper*, ce qui nous permettra de calculer le temps total lié à son exécution. Une fois une mesure effectuée, nous mettons ces données dans une `map` pour qu'ils soient accessibles depuis l'espace utilisateur.
Voici une version sommaire de la structure d'un programme à mesurer :

```
void foo() {
    __u64 t0, delta;
    t0 = bpf_ktime_get_ns();
    // Le code à mesurer se trouve ici
    delta = bpf_ktime_get_ns() - t0;
    // Ajout de delta à la map
}
```

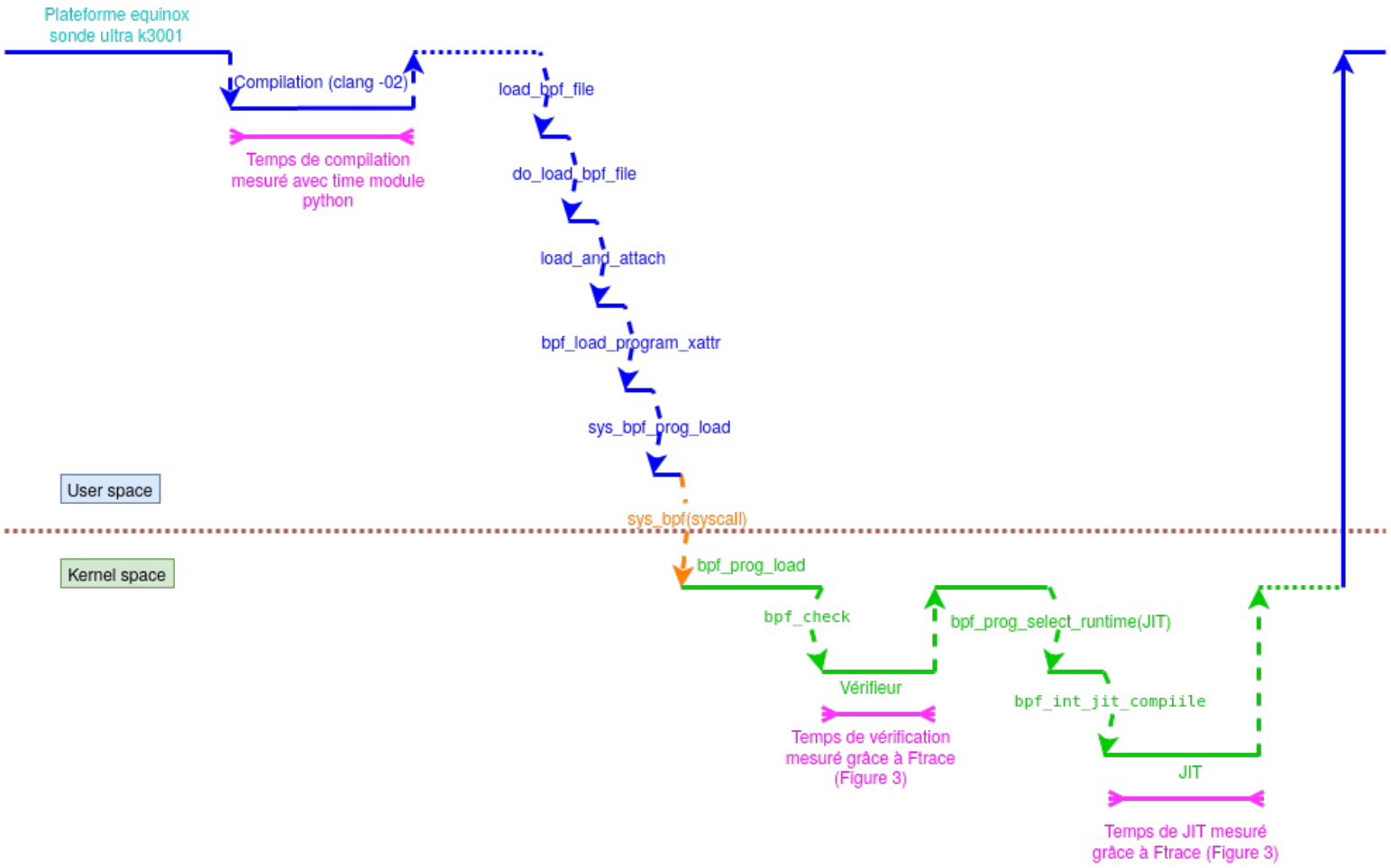

FIGURE 2 – Chronogramme de l'insertion d'un programme eBPF via notre plateforme.

En analysant le `bytecode` généré à l'aide de la commande `llvm-objdump` nous avons remarqué que dans certains cas (que nous n'avons pas réussi à identifier), dû au flag `-O2`, le `bytecode` généré pouvait se voir réordonné et avoir les deux appels à `bpf_ktime_get_ns` consécutifs, sans englober le code à mesurer. Ecrire les instructions en assembleur `eBPF` directement résout ce problème. Ce qui donne le code suivant :

```
void foo() {
    __u64 t0, delta, t1;
    __u64 *delta_addr = &delta, *t0_addr = &t0;
    asm volatile("call 5 \n r6 = r0" :::);
    // Le code à mesurer se trouve ici
    asm volatile("call 5 \n r0 -= r6" :::);
    asm volatile("*(u64 *)(%0 + 0) = r0"::"r"(delta_addr));
    // Ajout de delta à la map
}
```

Ici, `call 5` correspond à l'appel de `bpf_ktime_get_ns` et son résultat est stocké dans `r6` (qui est un registre de travail). Ensuite, le temps total est calculé et est stocké dans la variable `delta`. En utilisant cette méthode, nous n'avons observé aucun réordonnancement du `bytecode`.

4.2 Les baselines

Les programmes les plus proches du code natif système pouvant être chargés dynamiquement sont les modules noyau. Nous allons donc les utiliser comme points de comparaison pour évaluer l'impact de l'exécution du programme `eBPF` sur le système.

Compilation Pour la compilation des modules, nous allons mesurer le temps total comme pour les programmes `eBPF`. Les modules seront également compilés à l'aide d'un ficher `Makefile`. Nous utiliserons le même flag `-O2`, mais nous utiliserons le compilateur `gcc`. Cependant en prévention de compilation successive, lors de la construction, des fichiers sont générés pour optimiser la compilation, comme par exemple limiter les recherches de dépendance (avec les *.cmd).

Insertion L'insertion du module se fait via la commande `insmod` ou `modprobe`. Le `shell` exécutant cette commande effectue un `fork-exec` qui cherche et n'exécute que la fonction d'entrée définie via la macro `module_init(<module_init>)`. Le reste des fonctions, même si elles peuvent être appelées par le point d'entrée, peuvent être appelées par des événements extérieurs, hors du chargement, notamment par les technologies du `VFS`, de `kprobe`, de `tracepoint`, ...

Execution Pour la mesure de l'exécution des modules, nous allons utiliser la même structure que pour les programmes `eBPF`. La fonction utilisée dans les programmes `eBPF` est une fonction du noyau qui est exportée de manière à ce qu'elle soit accessible par les programmes sous licence `GPL`. Nous pourrons donc l'utiliser dans les modules. La structure ainsi que l'usage des mêmes fonctions nous permet d'avoir une symétrie entre les deux mesures, ce qui permet de minimiser les risques de biais liés aux façons de mesurer.
Voici une version sommaire de la structure d'un module à mesurer :

```
void foo() {
    __u64 t0, delta;
    __u32 i, res;
    t0 = ktime_get_mono_fast_ns();
    // Le code à mesurer se trouve ici
    delta = ktime_get_mono_fast_ns() - t0;
    // Ajout de delta au kobject
}
```

Comme nous pouvons le voir, les codes du programme `eBPF` et du module sont semblables. Les temps mesurés seront stockés dans un `kobject` pour les rendre accessibles depuis l'espace utilisateur.

4.3 Fonctionnement de la plateforme

Pour effectuer ces mesures, la plateforme a été développée avec le langage `Python 3.8.5`. Pour fonctionner, elle attend un fichier de configuration contenant :

- Pour le programme `eBPF` elle attend le chemin d'un dossier contenant : le fichier `C` du programme `eBPF`, un `Makefile` pour pouvoir le compiler.
- Pour le module, elle attend le chemin d'un dossier contenant le fichier `C` du module, un `Makefile` pour le compiler et le nom du `kobject` où les données temporelles pourront être récupérées.
- Le chemin d'un script à exécuter pour générer les événements liés aux programmes (appels systèmes, envois/réceptions de paquets, etc).

A partir de ce fichier, elle offre deux fonctionnalités :

Mesure du chargement Dans un premier temps, elle permet de mesurer le chargement des différents programmes. L'insertion sera réalisée 100 fois pour nous permettre de réaliser des graphiques à partir des données récupérées. Deux graphiques seront générés. Le premier représente le temps total d'insertion pour un programme `eBPF` comparé à l'insertion d'un module (appel à `insmod`). Le second concerne exclusivement `eBPF` et représente les temps d'exécution des deux fonctions `bpf_check` et `bpf_int_jit_compile` (cf Figure 4).

Mesure de l'overhead de l'exécution Dans un second temps, nous pouvons mesurer le temps d'exécution des deux types de programmes. En premier lieu, le programme`eBPF` est inséré dans le noyau(avec et sans `JIT` activé), ensuite la plateforme appelle le script qui génère les événements. Les temps d'exécution sont ensuite récupérés dans la `map`. Le même schéma est ensuite utilisé pour le module Linux. Chaque insertion est espacée d'un `sleep`, pour permettre des commutations afin de `flush` les caches, pour éviter de fausser les résultats avec des instructions déjà pré-chargées. Une fois ces deux étapes réalisées, un graphique sera généré comparant les deux types de programmes.

Pour ces deux mesures, les graphiques générés sont de type `boxplot` (boîte à moustache) couplé à un nuage de point pour représenter les données. Dans le cas de la mesure du surcoût, les points représentant la durée la plus élevée sont accompagnés de leur index d'exécution (de 0 à 99).

Le code de la plateforme et le jeu de test sont disponibles dans ce dépôt git.

5 Etude et résultats

Une fois la plateforme de test implémentée nous pouvons enfin comparer plusieurs programmes `eBPF` et des modules Linux afin de déterminer leur surcoût et en trouver l'origine.

Pour la réalisation des expériences, la configuration utilisée est composée d'une machine virtuelle s'exécutant avec le logiciel `virtuel QEMU emulator version 3.1.0`, avec un seul coeur mis à sa disposition. Elle exécute un noyau Linux `5.10.17`. La machine hôte exécute un noyau Linux `Mint 20.1 Cinnamon 4.8.6` avec un processeur `AMD Ryzen 5 3600 6-Core` et `15.6 GiB` de RAM. Les programmes `eBPF` sont compilés avec le compilateur `clang version 10.0.0-4ubuntu1` tandis que les modules utilisent `gcc (GCC) 10.2.0`.

5.1 Comparaison de l'insertion

Dans un premier temps nous allons comparer le temps d'insertion de programmes très simples.

micro-bench : Les progammes en question feront seulement un appel à : pour `eBPF`, la fonction `bpf_ktime_get_ns` et pour le module `ktime_get_mono_fast_ns`. Et ceci afin de limiter les optimisations du compilateur et d'avoir des temps suffisamment mesurables par les outils fournis par le système.

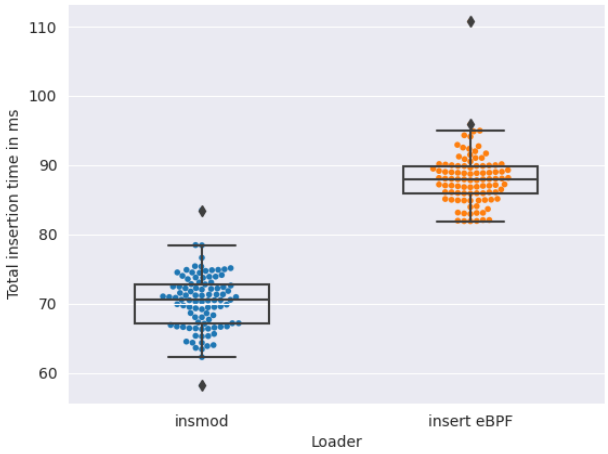

FIGURE 3 – Comparaison entre l'insertion d'un module noyau et d'un programme eBPF. Les temps sont exprimés en millisecondes.

Résultat des mesures : Nous remarquons sur la figure 3 qu'il y a une différence de temps entre l'insertion des deux technologies (d'un facteur `1,2` sur les médianes). Le temps mesuré pour l'insertion du module est le temps total du programme `insmod`, et pour le code `eBPF` c'est un programme qui fait simplement l'appel à la fonction `load_bpf_file`. Cette différence de temps s'explique par le fait que les modules Linux sont écrits en code natif et directement insérés dans le noyau sans

vérification tandis que les programmes `eBPF` subissent une vérification ainsi qu'une compilation supplémentaire avec le `JIT`.

Nous allons maintenant détailler l'analyse de l'insertion du programme `eBPF` qui est plus complexe que celle du module. Les temps sont exprimés en `µs`.

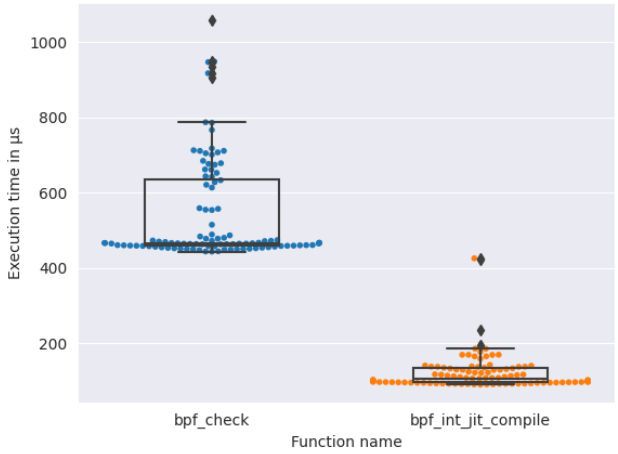

FIGURE 4 – Répartition de la charge entre le vérifieur et le compilateur Just-In-Time. Les temps sont exprimés en µs

Comme nous pouvons le voir dans la figure 4, dans le cas d'un programme simple, le temps du `JIT` et significativement inférieur à celui du `verifier`. Avec une médiane proche de `95µs` pour la fonction `bpf_int_jit_compile` et `460µs` pour la fonction `bpf_check` on se rend compte que le `JIT` a un impact moindre lors de l'insertion.

Ceci s'explique par la tâche complexe du `verifier` (`3.2.2`) qui doit vérifier l'intégrité et la sécurité du programme tandis que le `JIT` ne fait que compiler le `bytecode` qui a un jeu d'instructions faible.

Pour un temps d'insertion médian d'un programme `eBPF` total de `87 ms`, si nous additionnons les temps médians des deux fonctions mesurées, nous obtenons `455µs + 96µs = 551µs` soit `0.551ms`. Donc à elles deux, elles représentent `0,633%` du temps total. Cela s'explique par le fait que le programme est très simple et qu'il y a peu d'instructions à vérifier pour le `verifier` et peut à compiler pour le `JIT`.

Le temps restant devrait provenir des appels de fonctions successifs, des changements de contexte ou d'interruptions et des MISS de cache, ainsi que de l'appel système `bpf` (espace utilisateur vers noyau et inversement).

5.2 Mesure de l'exécution d'un programme simple

Nous allons maintenant mesurer le temps d'exécution pour un programme simple, ne réalisant pas de tâche particulière.

micro-bench : Dans cette partie nous allons nous intéresser au surcoût de l'exécution du programme mesuré dans la partie précédente. Pour rappel, les deux programmes utilisent la même fonction (`ktime_get_mono_fast_ns`) à la différence que le programme `eBPF` passe par un `helper` (`bpf_ktime_get_ns`) pour l'appeler.

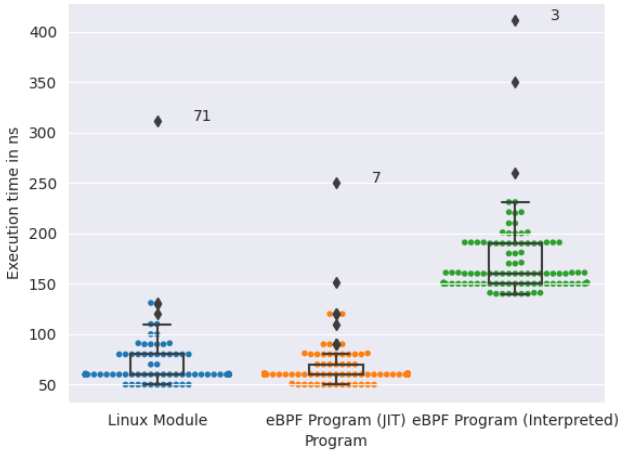

FIGURE 5 – Temps d'exécution du module Linux et du programme eBPF avec puis sans JIT exprimés en nanosecondes.

Résultat des mesures : La figure 5 représente les différents prélèvements de temps. Cette distribution en paliers n'est pas étonnante sachant que nous travaillons en `nanosecondes` et que la précision de la mesure est arrondie à la dizaine.

Comme nous pouvons le voir, les médianes pour un Module Linux et pour un programme `eBPF` exécuté avec le JIT sont égales (`60ns`) et la distribution des valeurs est également semblable pour les deux types de programmes. Cependant nous remarquons une différence significative pour un programme `ebpf` utilisant l'interpréteur (un facteur `2.7`). Ce à quoi on pouvait s'attendre d'un programme interprété. Cette fonctionnalité doit rester réservée aux architectures non supportées par le `JIT`. Le programme `eBPF` interprété a été mesuré en premier, c'est pour cela que l'index `3` qui correspond au premier appel de la fonction `ktime_get_mono_fast_ns`, celle-ci ne se trouvait probablement dans aucun niveau de cache. Les autres valeurs maximales peuvent être expliquées par un changement de contexte.

La similarité de ces résultats pour un module et un programme `eBPF` compilé avec le `JIT`, et l'écart avec un programme `ebpf` interprété montre également une cohérence sur notre métrique et sur notre

plateforme de test. En effet cette expérience revient à un appel de fonction pour notre module, et quant au programme `eBPF` avec JIT il ne fait qu'exécuter `ktime_get_mono_fast_ns`. Alors que le code `ebpf` interprété ne possède pas d'optimisation et doit passer par la couche de l'interpréteur pour effectuer sa tâche. Ce qui nous permet d'appréhender avec confiance les prochains tests.

5.3 Évaluation sur les boucles

Dans ce test nous allons tester le temps d'exécution d'une boucle dans eBPF et dans un module Linux.

Micro-bench : Le programme est constitué d'une boucle qui récupère le temps actuel avec la fonction $bpf\_ktime\_get\_ns()$ et fait la somme du temps, il est nécessaire d'appeler une fonction qui ne peut pas être pré-calculée pour éviter que le compilateur pré-calcule le résultat et enlève la boucle. Une fois le résultat calculé, nous enregistrons le temps d'exécution.

Résultat des mesures : Dans la figure 7 il est possible d'observer qu'il n'y a pas une différence considérable entre le module Linux et le programme $eBPF$ compilé avec le JIT. D'un autre côté, si nous incluons dans notre comparaison le programme eBPF qui n'a pas été compilé par le JIT (la figure 6), nous pouvons voir une différence importante entre les programmes compilés et les programmes interprétés. Le programme interprété a une médiane `8625ns` tandis que celle du programme compilé avec le `JIT` est de `7630ns`. Cela nous fait un ratio de `1,13` soit `13%` de surcoût. Ce coût devrait donc être pris en compte si nous sommes sur une architecture non supportée par le `JTI`.

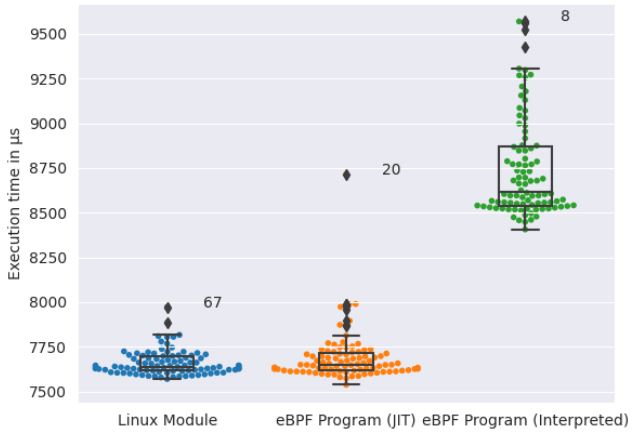

FIGURE 6 – Comparaison entre un module et un code eBPF (avec et sans JIT) sur boucle effectuant la même somme. Les temps sont exprimés en million de µs

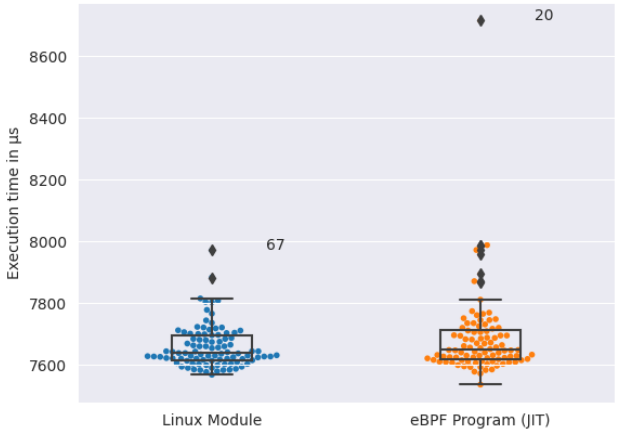

FIGURE 7 – Comparaison entre un module et un code eBPF avec JIT sur boucle effectuant la même somme. Les temps sont exprimés en million de µs

5.4 Mesure sur les déréférencements

Cette partie se concentre sur la mesure des coûts liés aux déréférencements de pointeurs.

micro-bench : Dans cette expérience nous allons comparer l'impact du déréférencement pour le stockage et la manipulation de variables dans une matrice stockée dans une `map`.

Résultat des mesures : Nous pouvons observer sur la figure 8 que le module possède une médiane d'environ `1,2 milliseconde`, ce qui est très proche de la médiane d'un code eBPF interprété qui avoisine les `1,35 milliseconde`. Alors qu'un programme eBPF exécuté avec JIT a une médiane inférieur à `0,2 milliseconde`.

Ce qui est très surprenant, et qui ne correspondait pas ce à quoi nous nous attendions. En cherchant une explication à une telle différence, nous avons remarqué une différence d'appel de fonction dans notre implémentation de notre module :

```
/* La nouvelle implémentation dans le module */
    j = get_random_u32();
/* Ancienne implémentation avec les données aberrantes */
    get_random_bytes(&j, sizeof(j));
```

Nous avons alors utilisé la même fonction dans notre module que celle utilisée dans le `eBPF`. Et nous obtenons le résultat suivant sur la figure 9.

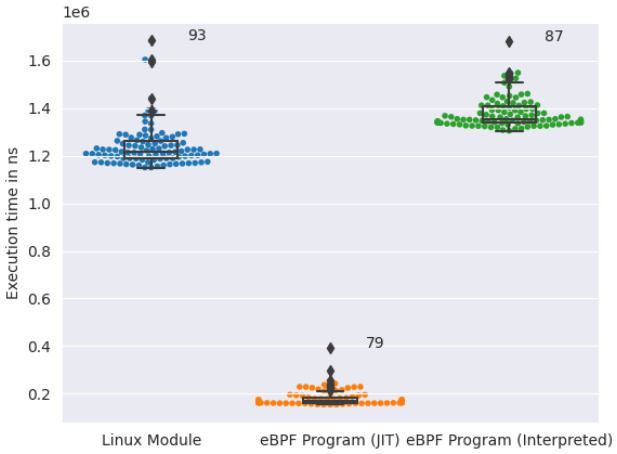

FIGURE 8 – Comparaison entre un module et un code eBPF(avec et sans JIT) sur des déréférencements de variables. Les temps sont exprimés en million de ns

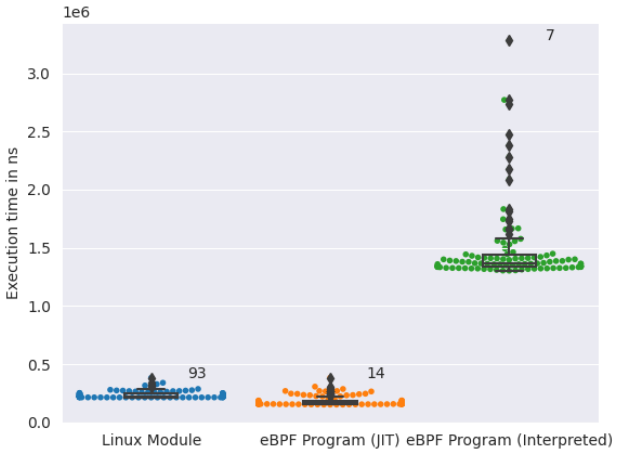

FIGURE 9 – Comparaison entre un module et un code eBPF(avec et sans JIT) sur des déréférencements de variables avec la même fonction de génération aléatoire. Les temps sont exprimés en million de ns

Nous obtenons ainsi un résultat plus cohérent. Cette différence provient du fait que la fonction `get_random_u32()` (de la famille des `get_random_int`) ne consomme que peu *pool* d'entropie, mais également qu'elle ne dispose pas de protection contre une lecture en arrière des valeurs aléatoires et enfin du fait que la graine de génération aléatoire n'est pas aussi souvent régénérée que la fonction `get_random_bytes()`.

La médiane du programme `eBPF` interprété étant relativement toujours au même niveau que précédemment, pour les mêmes raisons qu'évoquées dans la partie 5.2. Nous allons donc ignorer ce dernier et isoler les résultats pour le module et pour le code `eBPF` avec `JIT` dans la figure 10.

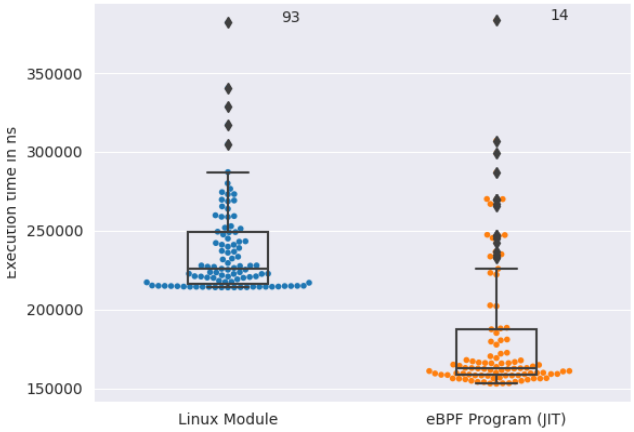

FIGURE 10 – Comparaison entre un module et un code eBPF avec JIT sur des déréférencements de variables avec la même fonction de génération aléatoire. Les temps sont exprimés en million de ns

Nous remarquons toujours une différence notable entre les deux types de programmes avec un ratio de `1,4`. Cette différence provient probablement du fait que `eBPF` possède des optimisations plus évoluées sur le parcours de liste et provoque moins de MISS de cache donnée.

Nous avions aussi pensé à l'éventualité que le déroulage des boucles sur `eBPF` pouvait lui procurer une certaine accélération. Toutefois en utilisant les mots clés suivants dans le module pour informer le compilateur de dérouler la boucle nous obtenons des résultats similaires. Il ne s'agit donc pas d'une accélération par déroulement de boucle (fait vérifié en désassemblant le code).

```
__attribute__((optimize("unroll-loops")))
```

5.5 Expérimentation sur le chaînage de programme

Les programmes `eBPF` permettent le chaînage de programmes entre eux. Dans cette partie nous allons mesurer le coût de ce chaînage et reproduire celui-ci avec un module Linux.

micro-bench : Pour mesurer le coût de l'appel en chaîne des programmes `eBPF`. Pour ce faire le programme `eBPF` utilise l'helper `bpf_tail_call`. Dans la version module noyau, nous utilisons une fonction qui est appelée récursivement. La profondeur de la récursion dans les deux cas sera limitée à 32, qui est la limite imposée par le noyau Linux pour les appels chaînés concernant les programmes `eBPF`.

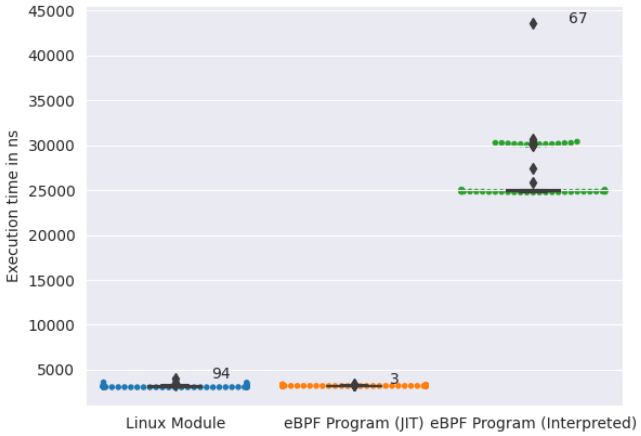

FIGURE 11 – Comparaison entre un module et un code eBPF (avec et sans JIT) sur un appel séquentiel de la même fonction. Les temps sont exprimés en million de ns

Résultat des mesures : Comme nous pouvons le voir sur la figure 11, le surcoût du programme `eBPF` interprété est significativement supérieur aux deux autres mesures. Le programme `eBPF` compilé `JIT` a une médiane de `3200ns` (figure 12) tandis que l'interprété a une médiane de `25 000ns`. Cela nous fait un ratio de `7.8`. Pour des architectures non supportées par le `JIT` d'`eBPF`, il peut être important de considérer cette différence.

Nous allons maintenant nous concentrer sur les différences entre le module du noyau Linux et du programme `eBPF`. Sur la figure 12, nous pouvons voir que le module Linux a un coût d'exécution plus faible que son pendant `eBPF`. Ils ont respectivement des médianes de `3125ns` et `3200ns`, pour un ratio de `1,024`. La différence n'est pas significative car les `tail calls` d'`eBPF` sont optimisés. En effet, cette fonctionnalité est implémentée comme un `long jump`. Seulement le pointeur d'instruction est modifié et le programme appelé utilisera la même pile que l'appelant.

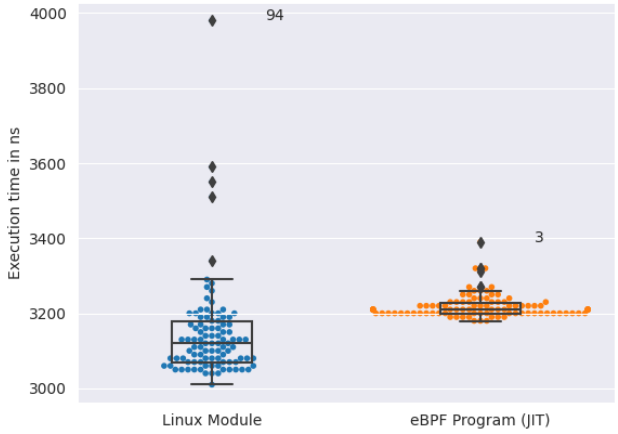

FIGURE 12 – Comparaison entre un module et un code eBPF avec JIT sur un appel séquentiel de la même fonction. Les temps sont exprimés en million de ns

6 Conclusion

Les programmes eBPF étaient à l'origine destinés pour le réseau mais sont actuellement utilisés de plus en plus dans l'industrie pour faire de l'introspection du noyau : débugguer, ajouter des sondes, comprendre l'exécution de celui-ci... En effet s'il s'est imposé c'est qu'il permet à la fois d'avoir un code sûr, rapide, fiable tout en étant facilement implémentable ce qui en fait un atout indéniable de nos jours. Cependant, malgré ces qualités, il n'existe pas à ce jour dans la littérature une réelle analyse de son impact sur le système ni même une métrique permettant de l'évaluer.

Pour répondre à cette problématique, nous nous sommes interrogés dans ce Projet SAR sur l'overhead de tels programmes eBPF. Pour cela nous avons établi comme baseline les modules Linux, étant donné qu'ils sont compilés vers du code natif, ils ont théoriquement de meilleures performances. S'en est suivi le développement de la plateforme de test, qui charge les modules Linux et les programmes eBPF, ensuite elle lance un nombre significatif de tests en générant des graphiques cohérents. Et pour cela nous avons eu besoin d'analyser toute la chaîne de compilation, d'insertion et d'exécution. Puis nous avons dû isoler des cas susceptibles d'être problématiques et produire en conséquence les codes eBPF avec leurs homologues en module. Et enfin désassembler le code compilé pour s'assurer que les optimisations appliquées n'aient pas affecté la fonctionnalité que nous cherchions à mesurer.

Grâce à notre démarche et à notre plateforme, nos résultats ont montré, dans un premier temps, que l'insertion d'un code simple avec un seul appel de fonction n'engendre pas de surcoût significatif. Ce qui nous a conforté sur la fiabilité de notre métrique et de notre plateforme de test. Une fois ce test simple effectué, nous avons pu implémenter des programmes permettant de tester des cas spécifiques qui pourraient générer des surcoûts significatifs. Nos mesures ont confirmé le fait que le code interprété était significativement plus lent que le code compilé avec le JIT. Mais il est tout de même important d'implémenter correctement les modules Linux car comme nous pouvons le voir dans la section 5.4, un mauvais choix de fonction peut rendre le module extrêmement plus lent que le programme eBPF.

D'après nos résultats, le surcoût d'un programme eBPF par rapport à un module est faible. Dans le cas du déréférencement, c'est le programme eBPF qui avait un surcoût plus faible. Cela peut être expliqué par les optimisations réalisées par clang. La différence entre les deux peut avoir son importance si elle concerne un programme qui s'exécute à une très haute fréquence. Dans tous les tests que nous avons faits nous n'arrivons pas a voir une différence importante entre les modules Linux et les programmes eBPF compilés par le JIT, par contre il y a une perte de performance importante quand le JIT est désactivé et le programme eBPF est interprété. Il est donc importante de vérifier si le matériel cible peut utiliser le JIT avant de décider d'utiliser un programme eBPF à la place d'un module Linux, si le programme eBPF doit être interprété, la perte de performance peut devenir inacceptable.

Perspectives

En plus des mesures que nous avons réalisées, il serait intéressant de mesurer des programmes ayant des applications réelles. Par exemple des programmes s'exécutant sur des serveurs supportant de grosses charges.

Sachant que tous nos tests ont été réalisés sur une machine virtuelle qui s'exécute avec un seul coeur de processeur de la machine hôte, il faudrait mesurer les mêmes programmes en augmentant le nombre de coeurs mis à sa disposition.

Il pourrait également être pertinent que notre plateforme soit capable de mesurer d'autres facteurs tels que l'empreinte mémoire d'un programme.

Les cartes programmables exécutant des programmes `eBPF` sur un processeur dédié se développent de plus en plus. Cela pourrait être judicieux d'en mesurer les performances. Cela impliquerait la modification des sondes de notre plateforme pour l'adapter à celles-ci.

Bibliographie

[1] S. Mccanne et V. Jacobson, « The BSD Packet Filter : A New Architecture for User-level Packet Capture, » 1992, p. 259-269.

[2] A. Starovoitov, *LKML : Alexei Starovoitov : [PATCH net-next] extended BPF*, https://lkml.org/lkml/2013/9/30/627, (Accessed on 03/08/2021), sept. 2013.

[3] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019, isbn : 978-0136554820. adresse : http://www.brendangregg.com/bpf-performance-tools-book.html.

[4] netoptimizer, *eBPF maps*, https://github.com/netoptimizer/prototype-kernel/blob/master/kernel/Documentation/bpf/ebpf_maps.rst, 2021.

[5] *bpf(2) - Linux manual page*, août 2019.

[6] IOVisorProject, *BCC*, https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md, 2021.

[7] *Linux Observability With BPF : Advanced Programming for Performance Analysis and Networking - Calavera, David, Fontana, Lorenzo - Livres*, https://www.amazon.fr/Linux-Observability-Bpf-Programming-Performance/dp/1492050202, (Accessed on 03/11/2021).

[8] ——, *bpftrace*, https://github.com/iovisor/bpftrace, 2021.

[9] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak et G. Carle, « Performance Implications of Packet Filtering with Linux eBPF, » in *2018 30th International Teletraffic Congress (ITC 30)*, t. 01, 2018, p. 209-217. doi : 10.1109/ITC30.2018.00039.

[10] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu et J. Pi, « Securing Linux with a Faster and Scalable Iptables, » *SIGCOMM Comput. Commun. Rev.*, t. 49, n° 3, p. 2-17, nov. 2019, issn : 0146-4833. doi : 10.1145/3371927.3371929. adresse : https://doi.org/10.1145/3371927.3371929.

[11] E. Reshetova, F. Bonazzi et N. Asokan, « Randomization Can't Stop BPF JIT Spray, » in *Network and System Security*, Z. Yan, R. Molva, W. Mazurczyk et R. Kantola, éd., Cham : Springer International Publishing, 2017, p. 233-247, isbn : 978-3-319-64701-2.

[12] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern et D. Miller, « The EXpress Data Path : Fast Programmable Packet Processing in the Operating System Kernel, » in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, sér. CoNEXT '18, Heraklion, Greece : Association for Computing Machinery, 2018, p. 54-66, isbn : 9781450360807. doi : 10.1145/3281411.3281443. adresse : https://doi.org/10.1145/3281411.3281443.