RFC 9669 : BPF Instruction Set Architecture (ISA)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 1 novembre 2024

Date de publication du RFC : Octobre 2024

https://www.bortzmeyer.org/9669.html

On a souvent envie de faire tourner des programmes à soi **dans** le noyau du système d'exploitation, par exemple à des fins de débogage ou d'observation du système. Cela soulève plein de problèmes (programmer dans le noyau est délicat) et la technique eBPF permet, depuis de nombreuses années, de le faire avec moins de risques. Ce RFC spécifie le jeu d'instructions eBPF. Programmeureuses en langage d'assemblage, ce RFC est pour vous.

eBPF désigne ici un jeu d'instructions (comme ARM ou RISC-V). Programmer en eBPF, c'est donc programmer en langage d'assemblage et, en général, on ne le fait pas soi-même, on écrit dans un langage de plus haut niveau (non spécifié ici mais c'est souvent un sous-ensemble de C) et on confie à un compilateur le soin de générer les instructions. Ce jeu d'instructions a plusieurs particularités. Notamment, il est délibérément limité, puisque toute bogue dans le noyau est particulièrement sérieuse, pouvant planter la machine ou pire, permettre son piratage. Vous ne pouvez pas faire de boucles générales, par exemple. eBPF est surtout répandu dans le monde Linux (et c'est là où vous trouverez beaucoup de ressources <https://www.kernel.org/doc/html/latest/bpf/index.html>) où il est une alternative aux modules chargés dans le noyau. Pas mal du code réseau d'Android est ainsi en eBPF. Normalisé ici, eBPF peut être mis en œuvre sur d'autres noyaux (il tourne sur Windows, par exemple). Le monde eBPF est très riche, il y a plein de logiciels (pas toujours faciles à utiliser), plein de tutoriels (pas toujours à jour et qui ne correspondent pas toujours à votre système d'exploitation) mais cet article se focalise sur le sujet du RFC : le jeu d'instructions.

On trouve de nombreux exemples d'utilisation en production par exemple le répartiteur de charge Katran <https://github.com/facebookincubator/katran> chez Facebook, via lequel vous êtes certainement passé, si vous utilisez Facebook. En plus expérimental, j'ai trouvé amusant qu'on puisse modifier les réponses DNS en eBPF <https://github.com/NLnetLabs/XDPeriments/tree/master/opt-extend>.

Passons tout de suite à la description de ce jeu d'instructions (ISA = "*Instruction Set Architecture*"). D'abord, les types (section 2.1) : `u32` est un entier non signé sur 32 bits, `s16`, un signé sur 16 bits, etc. eBPF

fournit des fonctions de conversions utiles (section 2.2) comme `be16` qui convertit en gros boutien (le RFC cite IEN137 <https://www.rfc-editor.org/ien/ien137.txt>...). Au passage, une mise en œuvre d'eBPF n'est pas obligée de tout fournir (section 2.4). La norme décrit des groupes de conformité et une implémentation d'eBPF doit lister quels groupes elle met en œuvre. Le groupe `base32` (qui n'a rien à voir avec le Base32 du RFC 4648 [1]) est le minimum requis dans tous les cas. Par exemple, `divmul32` ajoute multiplication et division. Tous ces groupes figurent dans un registre IANA <https://www.iana.org/assignments/bpf-instructions/bpf-instructions.xml#bpf-instruction-confo

Les instructions eBPF sont encodées en 64 ou 128 bits (section 3). On y trouve les instructions classiques de tout jeu, les opérations arithmétiques (comme `ADD`), logiques (comme `AND`), les sauts (`JA`, `JEQ` et autrs), qui se font toujours vers l'avant, pour, je suppose, ne pas permettre de boucles (souvenez-vous du problème de l'arrêt, qui n'a pas de solution avec un jeu d'instructions plus étendu), l'appel de fonction, etc.

En parlant de fonctions, eBPF ne peut pas appeler n'importe quelle fonction. Il y a deux sortes de fonctions utilisables, les fonctions d'aide (section 4.3.1), pré-définies par la plateforme utilisée, et non normalisées (pour celles de Linux, voir la documentation <https://man7.org/linux/man-pages/man7/bpf-helpers.7.html>, qui est sous `Documentation/bpf` si vous avez les sources du noyau). Il y a aussi les fonctions locales (section 4.3.2), définies par le programme eBPF.

Il y a enfin des instructions pour lire et écrire dans la mémoire (`LD`, `ST`, etc). Pour mémoriser plus facilement, eBPF utilise des dictionnaires ("*maps*", cf. section 5.4.1).

La section 6 concerne la sécurité, un point évidemment crucial puisque les programmes eBPF tournent dans le noyau, où les erreurs ne pardonnent pas. Un programme eBPF malveillant peut provoquer de nombreux dégâts. C'est pour cela que, sur Linux, seul root peut charger un tel programme dans le noyau. Le RFC recommande de faire tourner ces programmes dans un environnement limité (bac à sable), de limiter les ressources dont ils disposent et de faire tourner des vérifications sur le programme avant son exécution (par exemple, sur Linux, regardez cette documentation <https://www.kernel.org/doc/html/latest/bpf/verifier.html> ou bien l'article « "*Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions*" <https://dl.acm.org/doi/10.1145/3314221.3314590> »).

Enfin, section 7, les registres (pas les registres du processeur, ceux où on enregistre les codes utilisés). Deux registres IANA sont créés, celui des groupes de conformité <https://www.iana.org/assignments/bpf-instructions/bpf-instructions.xml#bpf-instruction-conformance-groups> et celui du jeu d'instructions <https://www.iana.org/assignments/bpf-instructions/bpf-instructio xml#bpf-instruction-set>. L'annexe A du RFC donne les valeurs actuelles. Les registres sont extensibles et la politique d'enregistrement est « Spécification nécessaire » et « Examen par un expert », cf. RFC 8126. (J'avoue ne pas savoir pourquoi, si les opcodes sont enregistrés, les mnémoniques ne le sont pas, cela rend les registres difficiles à lire.)

Un peu d'histoire, au passage. eBPF est dérivé de BPF, ce qui voulait dire "*Berkeley Packet Filter*", et était spécifique au filtrage des paquets réseau. Cet usage a été notamment popularisé par tcpdump. D'ailleurs, ce programme a une option pour afficher le code BPF produit :

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc4648.txt`

```
% sudo tcpdump -d port 53
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 10
(002) ldb      [20]
(003) jeq      #0x84            jt 6    jf 4
(004) jeq      #0x6             jt 6    jf 5
(005) jeq      #0x11            jt 6    jf 23
(006) ldh      [54]
...
(021) jeq      #0x35            jt 22   jf 23
(022) ret      #262144
(023) ret      #0
```

(Les mnémoniques `LDH` et `LDB` ne sont pas dans le RFC parce qu'ils sont des utilisations différentes du code `LD` ("*LoaD*"). Le H désigne un "*half-word*" - deux octets - et B un "*byte*". D'ailleurs, `RET` n'est pas dans le RFC non, plus, tcpdump est ancien et ne suit pas le RFC rigoureusement, il utilise ce qu'on nomme parfois cBPF, pour "*Classical BPF*".) Ensuite, BPF a évolué puis eBPF (qui n'est plus spécifique au filtrage de paquets et ce n'est donc plus un acronyme, juste eBPF tout court) dans Linux.

Si vous voulez vous mettre à eBPF (attention, la courbe d'apprentissage va être raide), `man 4 bpf` est utile. Typiquement, vous écrirez vos programmes dans un sous-ensemble de C et vous compilerez en eBPF, par exemple avec clang, après avoir installé tous les outils et bibliothèques nécessaires (il faut souvent des versions assez récentes) :

```
% cat count.c
...
int count_packets(struct __sk_buff *skb) {
    __u32 key = 0;
    __u64 *counter;

    counter = bpf_map_lookup_elem(&pkt_counter, &key);
    if (counter) {
        (*counter)++;
    }

    return 0;
}
...

% clang  -target bpf -c count.c

% file count.o
count.o: ELF 64-bit LSB relocatable, eBPF, version 1 (SYSV), not stripped

% objdump -d count.o
...
0000000000000000 <count_packets>:
   0: 7b 1a f8 ff 00 00 00 00  stxdw [%r10-8],%r1
   8: b7 01 00 00 00 00 00 00  mov %r1,0
  10: 63 1a f4 ff 00 00 00 00  stxw [%r10-12],%r1
  18: 18 01 00 00 00 00 00 00  lddw %r1,0
  20: 00 00 00 00 00 00 00 00
  28: bf a2 00 00 00 00 00 00  mov %r2,%r10
  30: 07 02 00 00 f4 ff ff ff  add %r2,-12
```

(Notez l'utilisation du désassembleur objdump.) Vous pouvez alors charger le code eBPF dans votre noyau, par exemple avec bpftool <https://github.com/libbpf/bpftool> (et souvent admirer de beaux messages d'erreur comme « *"libbpf : elf : legacy map definitions in 'maps' section are not supported by libbpf v1.0+"* »). Si tout fonctionne, votre code eBPF sera appelé par le noyau lors d'événements particuliers que vous avez indiqués (par exemple la création d'un processus, ou bien l'arrivée d'un paquet par le réseau) et fera alors ce que vous avez programmé. Comme me le fait remarquer Pierre Lebeaupin, il y a une bogue dans le source ci-dessus : l'incrémentation du compteur n'est pas atomique et donc, si on a plusieurs CPU, on risque de perdre certaines incrémentations. La solution de ce problème est laissé à la lectrice.

Un exemple d'utilisation d'eBPF pour observer ce que fait le noyau (ici avec un outil qui fait partie de bcc <https://github.com/iovisor/bcc>), on regarde les `exec` :

```
% sudo /usr/sbin/execsnoop-bpfcc
PCOMM            PID     PPID    RET ARGS
check_disk       389622  1628      0 /usr/lib/nagios/plugins/check_disk -c 10% -w 20% -X none -X tmpfs -X sy
check_disk       389623  1628      0 /usr/lib/nagios/plugins/check_disk -c 10% -w 20% -X none -X tmpfs -X sy
check_swap       389624  1628      0 /usr/lib/nagios/plugins/check_swap -c 25% -w 50%
check_procs      389625  1628      0 /usr/lib/nagios/plugins/check_procs -c 400 -w 250
ps               389627  389625    0 /bin/ps axwwo stat uid pid ppid vsz rss pcpu etime comm args
sh               389632  389631    0 /bin/sh -c   [ -x /usr/lib/php/sessionclean ] && if [ ! -d /run/systemd
sessionclean     389633  1         0 /usr/lib/php/sessionclean
sort             389635  389633    0 /usr/bin/sort -rn -t: -k2,2
phpquery         389638  389634    0 /usr/sbin/phpquery -V
expr             389639  389638    0 /usr/bin/expr 2 - 1
sort             389642  389638    0 /usr/bin/sort -rn
```

Le code eBPF est interprété par une machine virtuelle ou bien traduit à la volée en code natif.

Même ChatGPT peut écrire de l'eBPF <https://framagit.org/-/snippets/6937> (les tours de Hanoi et un serveur DNS).

De nombreux exemples se trouvent dans le répertoire `samples/bpf` des sources du noyau Linux. (Le fichier `README.rst` explique comment compiler mais seulement dans le cadre de la compilation d'un noyau. En gros, c'est `make menuconfig` , `cd samples/bpf` puis `make -i`.) Un bon exemple, relativement simple, pour commencer avec le réseau est `tcp_clamp_kern.c`.

Si vous préférez travailler en Go (là aussi, avec un Go récent…), il existe un bon projet <https://github.com/cilium/ebpf>. Si vous suivez bien la documentation <https://ebpf-go.dev/guides/getting-started/#compile-ebpf-c-and-generate-scaffolding-using-bpf2go>, vous pourrez compiler des programmes et les charger :

```
% go mod init ebpf-test
% go mod tidy
% go get github.com/cilium/ebpf/cmd/bpf2go
% go generate
% go build
% sudo ./ebpf-test
2024/08/20 15:21:43 Counting incoming packets on veth0..
...
2024/08/20 15:22:03 Received 25 packets
2024/08/20 15:22:04 Received 26 packets
2024/08/20 15:22:05 Received 27 packets
2024/08/20 15:22:06 Received 502 packets    <- ping -f
2024/08/20 15:22:07 Received 57683 packets
2024/08/20 15:22:08 Received 75237 packets
^C2024/08/20 15:22:09 Received signal, exiting..
```

Vous trouverez beaucoup de ressources eBPF sur . Et si vous voulez plonger dans les détails précis des choix de conception d'eBPF, je recommande ce document <https://www.kernel.org/doc/html/latest/bpf/bpf_design_QA.html>.

Ce RFC avait fait l'objet de pas mal de débats à l'IETF car, normalement, l'IETF ne normalise pas de langages de programmation ou de jeux d'instructions. (La première réunion était à l'IETF 116 en mars 2023 donc c'est quand même allé assez vite.)