Capítulo

3

Processamento Rápido de Pacotes com eBPF e XDP

Marcos A. M. Vieira, Racyus D. G. Pacífico, Matheus S. Castanho, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, Luiz F. M. Vieira

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG, Brasil
{mmvieira,racyus,matheus.castanho,elerson,epmcj,lfvieira}@dcc.ufmg.br

Abstract

Extended Berkeley Packet Filter (eBPF) has been rapidly adopted across multiple systems since its introduction in the Linux kernel in 2014. eBPF is used for fast packet processing. The use cases of eBPF have grown rapidly to include network monitoring, network traffic manipulation, load balancing, system monitoring, and so on. Several companies already use eBPF on projects such as Facebook, Netronome, and Cilium. This short-course aims to present eBPF. eBPF allows programming network devices. The developer can write in P4 or C language and then compile for eBPF instructions. The eBPF code can be processed in the Linux kernel or by programmable devices such as NetFPGAs and smartNICs. This short-course covers the main theoretical and fundamental aspects of eBPF, as well as introducing the reader to simple practical activities that can give insight into the general operation and use of eBPF.

Resumo

O filtro de pacotes estendido (Extended Berkeley Packet Filter (eBPF)) foi rapidamente adotado em vários sistemas desde sua introdução no kernel do Linux em 2014. O eBPF é utilizado para processamento rápido de pacotes. Os usos do eBPF cresceram rapidamente para incluir monitoramento de rede, manipulação de tráfego de rede, balanceamento de carga, monitoramento do sistema, etc. Várias empresas já utilizam eBPF em projetos como o Facebook, Netronome, Cilium. Este minicurso tem como objetivo apresentar o eBPF. O eBPF permite a programação dos dispositivos de redes. O desenvolvedor pode escrever em linguagem P4 ou C e depois compilar para instruções eBPF.

Depois, o código eBPF pode ser processado no kernel do Linux ou por dispositivos programáveis como NetFPGAs e smartNICs. O minicurso cobre os principais aspectos teóricos e fundamentais do eBPF, assim como introduzir o leitor a atividades práticas simples que possam dar uma visão sobre o funcionamento e uso geral do eBPF.

3.1. Introdução e Motivação

O constante aumento do tráfego na Internet e o crescimento da complexidade de serviços oferecidos em redes de centros de dados têm exigido taxas de processamento de pacotes cada vez mais altas. Além disso, a dinamicidade das demandas de serviços requer que a rede se adapte rapidamente para manter níveis adequados de qualidade de serviço e utilizar recursos disponíveis de forma eficiente. Porém, redes de computadores têm sido tradicionalmente desenvolvidas de forma estática, embutindo a implementação de protocolos de comunicação no *hardware* de dispositivos de rede, dificultando sua adequação às demandas atuais.

Nos últimos anos diversas propostas têm surgido visando adicionar maior programabilidade à rede. Dentre elas os paradigmas de SDN [Feamster et al. 2014] e NFV [Mijumbi et al. 2016], as linguagens POF [Song 2013] e P4 [Bosshart et al. 2014], e mais recentemente o filtro de pacotes estendido (*extended Berkeley Packet Filter* - eBPF) e o caminho de dados expresso (*eXpress Data Path* - XDP).

Este minicurso tem como objetivo apresentar o eBPF e o XDP para a comunidade brasileira de pesquisadores em redes de computadores. O eBPF é utilizado para modificar o processamento de pacotes no *kernel* do Linux e permite também a programação de dispositivos de rede. O desenvolvedor pode escrever uma aplicação na linguagem C ou P4 e depois compilá-la para instruções eBPF. O código eBPF resultante pode ser processado no *kernel* do Linux ou por dispositivos programáveis como NetFPGAs e placas de interface de rede inteligentes (*SmartNICs*).

O XDP fornece um caminho de dados de rede programável de alto desempenho no *kernel* do Linux. Ele efetua processamento de pacotes no ponto mais baixo da pilha de software, o que o torna ideal para aplicações de alta velocidade sem comprometer a programabilidade. Além disso, novas funções podem ser implementadas dinamicamente com o caminho rápido integrado sem modificação do código fonte do *kernel*. O programa eBPF modifica o funcionamento do *kernel* (programável), porém não precisa recompilá-lo pra isso.

A importância do eBPF e XDP é percebida pela sua rápida adoção desde sua introdução no *kernel* do Linux em 2014, tanto pela indústria e quanto por projetos de pesquisa na academia. Seus casos de uso cresceram rapidamente para incluir tarefas como monitoramento de rede, manipulação de tráfego de rede, balanceamento de carga e introspecção do sistema operacional. Várias empresas já utilizam eBPF em projetos, como Facebook [Facebook 2018], Netronome [Beckett et al. 2018] e Cloudflare [Bertin 2017].

O minicurso está organizado da seguinte forma: o restante desta seção apresenta uma introdução ao BPF. Na seção 3.2, é descrita a arquitetura da máquina eBPF. A seção 3.3 explica a visão geral do sistema eBPF. Na seção 3.4, é definido o que são ganchos (*hooks*) e onde eles ocorrem na pilha de rede. Também é explicado o que é XDP e como

utilizá-lo. A seção 3.5 descreve aspectos dos programas eBPFs, como os tipos de programas disponíveis, o que são mapas, os tipos de mapas disponíveis e como utilizá-los em um programa eBPF. Também define-se o que são funções auxiliares e tem-se a descrição das presentes no *kernel* do Linux. Na seção 3.6 são apresentadas algumas ferramentas úteis para o desenvolvimento e depuração de programas eBPF. A seção 3.7 descreve as plataformas de software e hardware já existentes capazes de processar instruções eBPF. Na seção 3.8 são apresentadas exemplos de funções de rede que utilizam o eBPF. A seção 3.9 discute vários projetos de pesquisas importantes já desenvolvidos e a seção 3.10 apresenta as atuais limitações da arquitetura e dá sugestões de como superá-las. Finalmente, a seção 3.11, conclui o minicurso. Todo o código utilizado neste minicurso está disponível no [Vieira et al. 2019].

3.1.1. O que é BPF

O filtro de pacotes BPF (*Berkeley Packet Filter*) [McCanne and Jacobson 1993] foi proposto por Steven McCanne and Van Jacobson em 1992, como uma solução para realizar filtragem de pacotes no *kernel* de sistemas Unix BSD. Ele consistia em um conjunto de instruções e uma máquina virtual (VM) para execução de programas escritos nessa linguagem.

Inicialmente, o código de bytes (bytecode) de uma aplicação era transferido do espaço de usuário para o *kernel*, onde era então verificado para prevenir problemas de segurança e falhas no *kernel*. Após passar na verificação, o programa era anexado a um soquete e executado a cada pacote recebido. Essa habilidade de executar programas fornecidos pelo usuário no *kernel* de forma segura se mostrou uma boa escolha de design do BPF.

Outro fator de destaque do BPF é seu conjunto de instruções simples e bem definido. Aliado à existência de uma máquina de compilação JIT (*Just-In-Time*) para BPF no *kernel*, esses fatores foram fundamentais para o bom desempenho da ferramenta.

Além da definição do código de bytes, o BPF também define um modelo de memória baseado em pacotes (instruções de carga são implicitamente feitas no pacote de processamento), registradores (A e X, acumulador e registrador de índice), um armazenamento de memória temporário e um contador de programa implícito. O lado esquerdo da figura 3.1 (Máquina BPF Clássica) ilustra a arquitetura da máquina BPF.

O *kernel* do Linux possui suporte ao BPF desde a versão 2.5. Não houve grandes alterações no código BPF até 2011, quando o interpretador BPF foi modificado para ser um tradutor dinâmico [Dumazet 2011]. Em vez de interpretar o código de bytes BPF, agora o *kernel* era capaz de traduzir os programas BPF diretamente para uma arquitetura de destino: x86, ARM, MIPS, etc.

Um dos usuários mais proeminentes do BPF é a biblioteca *libpcap*, presente na ferramenta *tcpdump*. Ao utilizar o *tcpdump* para capturar pacotes, um usuário pode definir uma expressão de filtragem de pacotes de modo que somente pacotes que correspondem a essa expressão serão realmente capturados. Por exemplo, a expressão "*ipv4 tcp*" captura todos os pacotes IPv4 que contem o protocolo da camada de transporte TCP. Essa expressão pode ser reduzida por um compilador para o código de bytes BPF. O código 1

apresenta um exemplo de programa BPF para filtrar pacotes permitindo capturar apenas segmentos TCP.

Código 1 Exemplo de program BPF para permitir apenas segmentos TCP.

```
(000) ldh [12]
(001) jeq #0x800 jt 2 jf 5
(002) ldb [23]
(003) jeq #6  jt 4 jf 5
(004) ret #-1
(005) ret #0
```

Basicamente, o que o código 1 faz é:

- Instrução (000): carrega (*load (ld)*) o *offset* 12 do quadro, como uma palavra de 16 bits, no acumulador. O deslocamento 12 representa o tipo de pacote no quadro Ethernet.
- Instrução (001): compara o valor do acumulador com 0x800, que é o valor *EtherType* do IPv4. Se o resultado for verdadeiro, o contador do programa salta (*jt jump true - desvio se verdadeiro*) para a instrução (002), caso contrário salta *jf jump falso - desvio se falso* para a instrução (005).
- Instrução (002): carrega o *offset* 23 do quadro, como um byte, no acumulador. O deslocamento 23 representa o campo protocolo do pacote IPv4. A contagem é a partir do início do quadro Ethernet.
- Instrução (003): compara o valor com 6 (valor do campo protocolo do pacote IPv4 que indica se contem um segmento TCP). Se for verdadeiro, pular para a instrução (004), se não ir para a instrução (005).

O programa de filtragem de pacotes executa até retornar um resultado, que geralmente é um booleano. Retornar um valor diferente de zero (instrução (004)) significa que o pacote casou com o filtro, enquanto que retornar um valor zero (instrução (005)) significa que o pacote não corresponde com o filtro e por isso será descartado.

3.2. Máquina eBPF

Ao longo do tempo, alguns aspectos de design do BPF não conseguiram se sustentar tão bem. O design da VM e da arquitetura do conjunto de instruções (ISA) ficaram defasados na medida que os processadores modernos migraram para registradores de 64 bits e novas instruções necessárias para sistemas multiprocessadores foram adicionadas. Com isso, o foco do BPF em prover um número pequeno de instruções RISC deixou de corresponder às realidades dos processadores modernos.

Dessa forma, uma nova versão BPF foi introduzida para aproveitar os novos recursos dos hardwares modernos. Essa nova versão foi batizada de eBPF, enquanto a anterior se tornou o cBPF (BPF clássico) e foi introduzida na versão 3.15 do *kernel*.

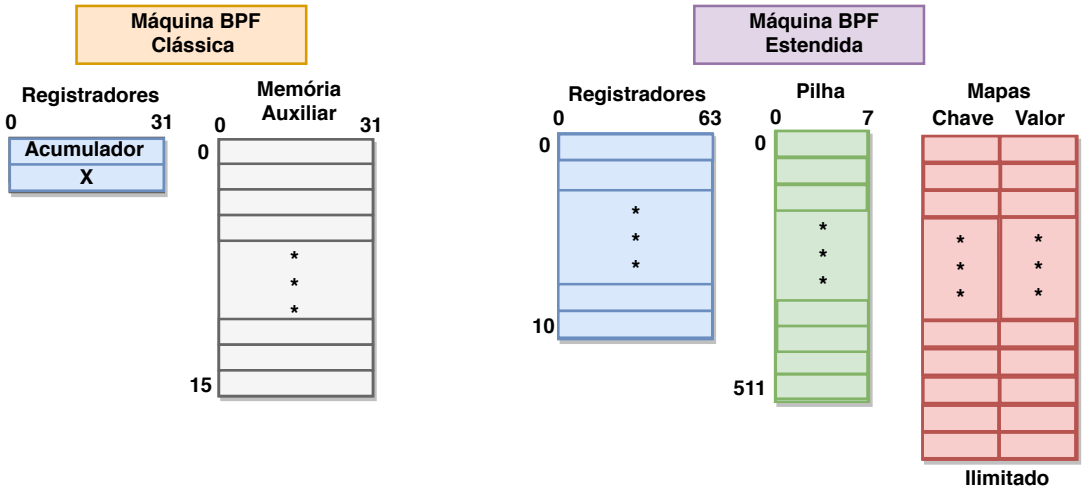


Figura 3.1. Processador BPF e eBPF.

O lado direito da figura 3.1 ilustra a máquina eBPF. O número de registradores aumentou de 2 para 11 (dos quais 10 registradores podem ser escritos), a largura dos registradores passou de 32 bits para 64 bits, o conjunto de instruções passou a ser de 64 bits e a máquina agora possui uma pilha de 512 bytes. Também foram adicionados armazenamentos de dados globais, denominados mapas, e a opção de chamadas de funções que são executadas dentro do *kernel*, denominadas funções auxiliares [Schulist et al. 2019].

O eBPF também adicionou o recurso de chamadas de cauda (*tail-calls*) para contornar a limitação de tamanho máximo de 4096 bytes dos programas eBPF. Com ele, um programa eBPF é capaz de passar o controle de execução para um novo programa eBPF.

No cBPF era necessário definir os desvios para os casos verdadeiros e falsos em um programa. Já no eBPF só é necessário definir os desvios verdadeiros, sendo que os desvios falsos seguem a sequência de execução do programa (chamados *jump fall-through*).

O eBPF também adicionou novas instruções ao conjunto de instruções. Entre as inclusões estão instruções aritméticas e lógicas para os registradores, bem como uma instrução para chamadas de função no *kernel* de forma barata. O eBPF adota a mesma convenção de chamada da linguagem C. Parâmetros são passados para funções através de registradores da máquina virtual, assim como acontece nativamente no hardware. Isso permite mapear uma instrução de chamada de função eBPF para uma única instrução de chamada nativa, habilitando a chamada da função quase sem custo. Esta facilidade é utilizada pelo eBPF para suportar as funções auxiliares, permitindo que os programas eBPF interajam com o *kernel* durante o processamento e realizem chamadas de sistema.

A máquina virtual eBPF suporta o carregamento dinâmico e o recarregamento de programas. O *kernel* gerencia o ciclo de vida de todos os programas. Isso permite estender ou limitar a quantidade de processamento realizado para uma determinada situação, adicionando ou removendo partes do programa que não são necessárias e recarregá-las novamente.

3.2.1. Classes de instruções e conjunto de instruções

O eBPF possui sete classes de instruções: *load* imediato (BPF_LD), *load* estendido para 64 bits (BPF_LDX), *store* imediato (BPF_ST), *store* estendido para 64 bits (BPF_STX), operações lógicas e aritméticas (BPF_ALU), desvios (BPF_JMP), operações lógicas e aritméticas de 64 bits (BPF_ALU64).

BPF_LD, BPF_LDX: Ambas as classes são para operações de carregamento. O BPF_LD é usado para carregar uma palavra dupla como uma instrução especial utilizando duas instruções devido ao imediato ter apenas 32 bits e precisar de 64 bits. Também é usado para carregar 8 bits (byte (B)), 16 bits (meia palavra *half word*(H)), 32 bits (palavra *word*(w)), e 64 bits (palavra dupla *double word* (DW)). A classe BPF_LDX contém instruções para carregamentos de byte / meia palavra / palavra / palavra dupla fora da memória. A memória neste contexto é genérica e pode ser memória de pilha, dados de valor de mapa, dados de pacote, etc.

BPF_ST, BPF_STX: Ambas as classes são para operações de armazenamento. Semelhante a BPF_LDX, o BPF_STX é a extensão de BPF_ST e é usado para armazenar os dados de um registrador na memória, que, novamente, pode ser da pilha, valor de mapa, dados de pacote, etc. BPF_STX também contém instruções especiais para executar palavras e palavras duplas baseados em operações de adição atômica, que podem ser usadas para contadores, por exemplo. A classe BPF_ST é semelhante a BPF_STX fornecendo instruções para armazenar dados na memória apenas que o operando de origem é um valor imediato.

BPF_ALU, BPF_ALU64: Ambas as classes contêm operações da ULA (Unidade Lógica e Aritmética). Geralmente, as operações BPF_ALU estão no modo de 32 bits e BPF_ALU64 no modo de 64 bits. Ambas as classes da ULA possuem operações básicas com o operando de origem, que é baseado em registro e uma contraparte baseada em imediato. As operações suportadas por ambos são: soma, subtração, e lógico, ou lógico, deslocamento à esquerda («), deslocamento à direita (»), ou exclusivo, multiplicação, divisão, resto e negação. Também a instrução mover (*mov*) foi adicionada como uma operação especial da ULA para ambas as classes nos dois modos de operandos. BPF_ALU64 também contém um deslocamento à direita com sinal. Além disso, BPF_ALU contém instruções de conversão de *endianness* para meia palavra, palavra, palavra dupla em um dado registro de fonte.

BPF_JMP: Esta classe é dedicada a operações de salto. Os saltos podem ser incondicionais e condicionais. Saltos incondicionais simplesmente movem o contador de programa para frente, de modo que a próxima instrução a ser executada em relação à instrução atual seja *deslocamento + 1*, onde o deslocamento é codificado na instrução. Como deslocamento tem sinal (*signed)*, o salto também pode ser executado para trás, desde que esteja dentro dos limites do programa. Os saltos condicionais operam em operandos de origem baseados em registro e baseados em imediato. Se a condição nas operações de salto resultar em verdadeiro, então um salto relativo para desvio + 1 é realizado, caso contrário, a próxima instrução (0 + 1) é executada. Essa lógica de salto é diferente em comparação com o BPF e permite uma melhor previsão de ramificação, pois ela se ajusta à lógica do preditor de desvios da CPU com mais naturalidade. As condições disponíveis são igualdade (jeq ==), diferença (jne !=), maior (jgt >), maior ou igual (jge >=), maior

com sinal (jsgt >), maior ou igual com sinal (jsge >=), menor (jlt <), menor ou igual (jle <=), menor com sinal (jslt <), menor ou igual com sinal (jsle <=) e jset (salto se origem e destino). Além disso, há três operações de salto especiais dentro dessa classe: a instrução de saída (*exit*) que deixará o programa BPF e retornará o valor atual em r0 como um código de retorno, a instrução de chamada *call*, que emitirá uma chamada de função em uma das funções auxiliares do eBPF e uma instrução de chamada cauda *tail*, que saltará para um outro programa eBPF.

3.2.2. Código de byte do eBPF

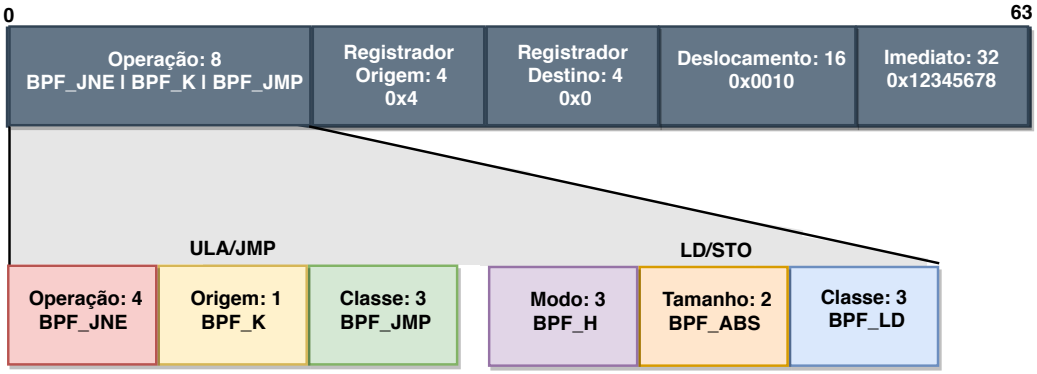


Figura 3.2. Código de byte eBPF.

A figura 3.2 mostra o código de byte do eBPF. O processador eBPF possui instruções de 64 bits, sendo 32 bits para representação de um número imediato (`imm`), 16 bits para o `offset`, 4 bits para registrador de origem, 4 bits para registrador de destino e, por fim, o código da operação (`opcode`) de 8 bits. O opcode pode ser redividido dependendo da classe da instrução. Para *load (LD)* e *store (STO)*, o opcode possui 3 bits mais significativos para modo de acesso à memória, 2 bits para tamanho da palavra e 3 bits para classe de instrução. Para as instruções de desvio *jump (JMP)*, e que utilizam a unidade lógica e aritmética (ULA), os 4 bits mais significativos especificam qual é a operação, 1 bit para especificar se a instrução é realizada com o valor imediato ou com o operando de origem e por fim 3 bits menos significativos para a classe da instrução.

3.2.3. Conjunto de Registradores

A tabela 3.1 descreve a funcionalidade de cada registradores eBPF. O registrador r10 é apenas de leitura e serve para armazenar o ponteiro do quadro para acesso à pilha. O registrador r0 é onde armazena o valor de retorno de função. É nesse registrador que fica armazeno ao final da computação qual a ação que será feita no encaminhamento do pacote.

3.3. Sistema eBPF

O sistema eBPF é composto por uma série de componentes responsáveis por compilar, verificar e executar códigos fonte de aplicações desenvolvidas. Mais detalhes sobre eles são dados a seguir.

Tabela 3.1. Descrição do conjunto de registradores do eBPF

Registrador	Descrição
R0	valor de retorno de funções e da saída do programa eBPF.
R1 - R5	passagem de argumento de funções.
R6 - R9	registradores que preservam valores em chamadas de função.
R10	ponteiro do quadro para acesso a pilha. É apenas de leitura.

3.3.1. Visão Geral

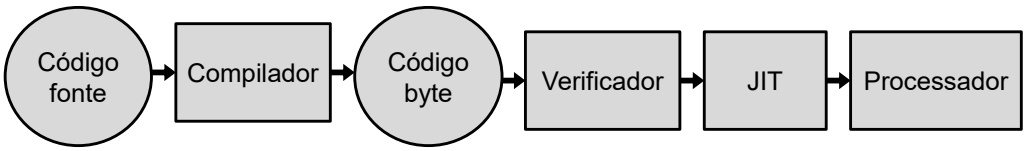


Figura 3.3. Fluxo de Trabalho para o eBPF.

O fluxo de trabalho típico do sistema eBPF é ilustrado na figura 3.3. Os programas eBPF são escritos em linguagem de alto nível (C, P4, Go), compilados pelo LLVM em arquivos objeto/ELF, que são analisados pelo carregador BPF ELF do espaço do usuário (como *iproute2* ou outros) e inseridos no *kernel* por meio da chamada do sistema BPF. O *kernel* verifica as instruções eBPF e faz a tradução dinâmica (JIT), retornando um novo descritor de arquivo para o programa. Este descritor pode então ser anexado a um subsistema (por exemplo, rede). Se suportado, o subsistema poderia, então, transferir o programa BPF para o hardware (por exemplo, a placa de rede) senão é executado pelo próprio processador.

3.3.2. Compilador

Linguagens de alto nível podem ser utilizadas para escrever código para o plano de dados que pode ser compilado para o conjunto de instruções do eBPF. O compilador LLVM 3.9 possui um *backend* para a plataforma eBPF, permitindo programar em um subconjunto de C e gerando código executável no formato eBPF.

Também é possível gerar instruções eBPF através de linguagens de domínio específico, como por exemplo, P4 [Bosshart et al. 2014]. Existem esforços do projeto de código aberto IOVisor [IOvisor 2019] e da VMWare [VMWare 2018] que já implementaram um compilador de P4 para eBPF [Budiu 2015].

3.3.2.1. Subconjunto de C

É possível programar para eBPF usando um subconjunto da linguagem C. Esse subconjunto exclui algumas bibliotecas externas, chamadas de sistemas e aritmética de ponteiro enquanto provê funções para definição e manipulação de tabelas.

Existem alguns detalhes importantes:

- O eBPF só pode utilizar um subconjunto de bibliotecas da linguagem C. Por exem-

plo, a função *printf()* não está disponível para a utilização. Neste caso, é necessário definir e utilizar funções auxiliares.

- Ainda não se tem noção de chamada de funções. Logo, todas as chamadas de funções são *inline*. Neste caso, basta declara a função como *inline*.
- Não se pode ter nenhuma variável global. A solução é utilizar mapas.
- Devido ao verificador, (ainda) não se pode ter laços (*loops*) a menos que sejam desenrolados através da diretiva de compilação *#pragma clang loop unroll (full)*. Outra solução é desenvolver um sistema eBPF mas que não inclua o verificador.
- Ainda não se tem suporte a cadeia de caracteres (*strings*).
- O espaço de pilha é limitado até 512 bytes.

3.3.3. Verificador

Para garantir a integridade e segurança do sistema operacional, o *kernel* do Linux utiliza um verificador que faz a análise estática de programas com instruções eBPF sendo carregados no sistema. A sua implementação é feita no arquivo `kernel/bpf/verifier.c`.

O verificador permite checar a validade, segurança e desempenho dos programas eBPF. Entre outras coisas, ele verifica se um programa termina, se os acessos de memória estão nos intervalo de memória permitidos para o programa e qual a maior profundidade do caminho de execução. O verificador é chamado depois do código ter sido compilado e antes de carregá-lo no plano de dados. Uma boa descrição geral é apresentada em [Høiland-Jørgensen et al. 2018].

A verificação da condição de término do programa é feita pelo verificador através da geração de um grafo acíclico direcionado (GAD). Programas eBPF que não possuem saltos para trás ou que possuem apenas laços de tamanho pré-definidos (podendo assim ter o laço desenrolado (*loop unroll*)) podem ser sintetizado em um GAD e isso garante o término deles. Um exemplo de geração de GAD é dado a seguir.

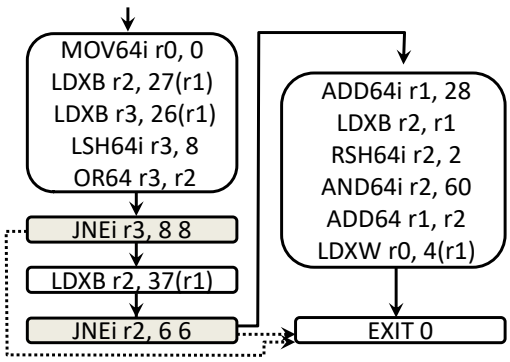


Figura 3.4. Exemplo de Grafo Acíclico Direcionado para o código eBPF.

O código 2 ilustra um exemplo de código na linguagem C para acessar o campo número de sequência do protocolo TCP. A figura 3.4 mostra o respectivo GAD para o

Código 2 Exemplo de código em C para acessar o número de sequência TCP.

```c
#include <linux/if_ether.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include "ebpf_switch.h"

uint64_t prog(struct packet *pkt){
  if (pkt->metadata.in_port == 0) {

    // Verifica se o quadro Ethernet possui um pacote ipv4
    if (pkt->eth.h_proto == 0x0008) {
      struct ip *ipv4 = (struct ip *)
        ( ((uint8_t *)&pkt->eth) + ETH_HLEN );

      // Verifa se pacote IPv4 possui segmento TCP
      if (ipv4->ip_p == 6) {
        struct tcphdr *tcp = (struct tcphdr *)
          ( ((uint32_t *) ipv4)+ ipv4->ip_hl );
     // retorna número de sequência
        return tcp->th_seq;
      }
    }
  }
  return 0;
}
```

código presente no código 2. Cada nó do GAD contém uma ou mais instruções do eBPF. Instruções terminando com a letra i indicam o uso de valores imediatos.

Os nós de saltos condicionais são aqules que contêm duas linhas de saída e plano de fundo cinza claro. Linha sólida indica o próximo nó do ACFG. Linha pontilhada indica um salto para outro nó do ACFG. No exemplo dado, os nós de salto condicionais contêm a instrução desvio não igual (*jump not equal (jnei)*). O último valor da instrução jnei indica o número instruções para pular quando a condição é válida.

Para entender melhor o programa eBPF, vale lembrar que o registrador R1 começa com um ponteiro para o pacote armazenado na memória de dados e o registrador R0 armazena o valor de retorno, conforme descrito na tabela 3.1.

No primeiro nó, as instruções de byte de carga extraem os bytes 26 e 27 da memória de dados. Dado que os metadados têm 16 bytes, isso representa os bytes 10 e 11 do pacote (contando a partir de 0), que é o campo do tipo Ethernet. Em seguida, uma troca de bytes é feita para definir a ordem dos bytes (*endianess*). A instrução de primeiro salto compara se o tipo de Ethernet é 0x0800. Em seguida, o campo do protocolo IP é extraído do pacote IPv4 e verifica-se se é o protocolo TCP (valor 6). Finalmente, o número de sequência do TCP é extraído e colocado no registrador de retorno (R0). A última instrução informa que o código termina. O GAD é útil para determinar o pior tempo de execução do caso.

3.3.4. Pseudo-sistema de arquivos

O sistema BPF apresenta um pseudo-sistema de arquivos localizado em `/sys/fs/bpf`. Através dele é possível ter acesso aos mapas (seção 3.5.2) de programas BPF carregados no sistema.

3.4. Ganchos (Hooks)

Gancho (*Hook*) é uma interface que permite que um programador insira programação customizada no *kernel*. O gancho permite modificar ou melhorar o comportamento de um sistema operacional, aplicações ou outros componentes de software através da interceptação de chamadas de funções, mensagens ou eventos passados entre componentes de software.

Em Redes de Computadores, ganchos são utilizados para a interceptação de pacotes antes da chamada ou durante a execução no sistema operacional. O processamento da pilha de rede do sistema operacional pode ser custoso. O gancho permite que aplicações que executam no espaço de usuário possam processar pacotes evitando passar nessa pilha e, assim, diminuir o custo adicional de processar pacotes.

3.4.1. Tipos de ganchos

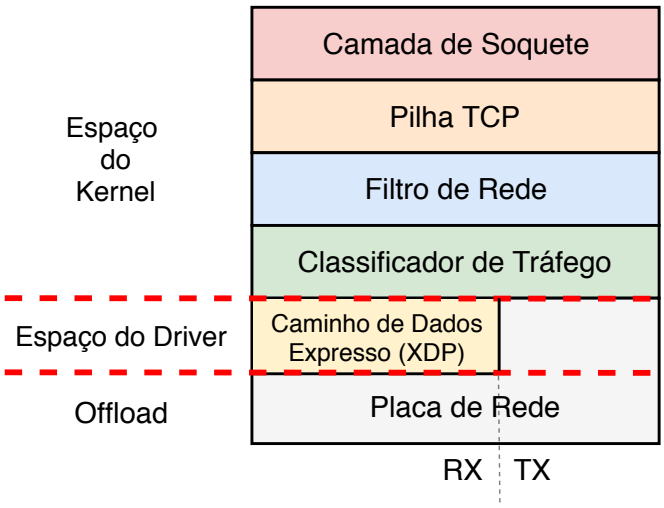


Figura 3.5. Camadas para ganchos.

Os pacotes de entrada são transmitidos através das camadas do *kernel* da placa de rede, conforme mostrado na figura 3.5. As camadas são: camada de soquete, pilha TCP, filtro de rede (*Netfilter*), classificador de tráfego (*Traffic Control/Classifier (TC)*), caminho de dados expresso (*eXpress Data Path (XDP)*) e a placa de rede.

Os pacotes podem ser destinados a um aplicativo do espaço do usuário ou podem ser interceptados por um módulo do *kernel*, como `iptables`, localizados na camada de filtro da rede. Os programas eBPF podem se conectar a vários locais dentro do *kernel* do Linux e filtrar pacotes. O melhor desempenho pode ser obtido filtrando os pacotes nas camadas inferiores do *kernel*, resultando no gancho XDP ser a melhor opção em termos de desempenho.

3.4.2. O que é XDP

O gancho Caminho de Dados Expresso (*eXpress Data Path (XDP)*) ocorre dentro do driver do dispositivo de rede, permitindo que os pacotes sejam processados pela CPU no primeiro ponto. O gancho XDP também permite transferir a computação para a placa de rede.

3.4.3. Ações XDP

A tabela 3.2 descreve as ações do XDP, incluindo os valores, nome e descrição. Os pacotes serão encaminhados de acordo com o valor da ação. O valor de retorno do programa eBPF é armazenado no registrador 0. Esse valor de retorno determina qual é a ação XDP correspondente.

Tabela 3.2. Descrição do conjunto de ações do XDP

Valor	Ação	Descrição
0	XDP_ABORTED	Erro. Descarta o pacote.
1	XDP_DROP	Descarta o pacote.
2	XDP_PASS	Permite o pacote continuar até o *kernel*.
3	XDP_TX	Devolve o pacote para a rede.
4	XDP_REDIRECT	Redireciona o pacote.

As quatro primeiras ações são de retorno simples (sem parâmetros), que podem abortar (descarta o pacote e passa o *tracepoint trace_xdp_exception*), descartar pacote, permitir que ele seja processado pela pilha de rede do *kernel*, ou imediatamente retransmiti-lo pela mesma interface de rede. O XDP_REDIRECT permite que o programa XDP redirecione o pacote, oferecendo controle adicional sobre seu processamento posterior.

A ação de redirecionamento requer um parâmetro adicional que especifica o alvo de redirecionamento, que é definido através de uma função auxiliar antes do programa sair. A funcionalidade de redirecionamento pode ser usada (i) para transmitir o pacote em uma interface de rede diferente (incluindo interfaces virtuais conectadas para máquinas virtuais), (ii) para passá-lo para uma CPU diferente para processamento adicional, ou (iii) para passá-lo diretamente para um soquete (AF_XDP) no espaço de usuário. Além disso, como o parâmetro de redirecionamento é implementado como uma pesquisa de mapa (onde o programa XDP fornece a chave de pesquisa), as regras de redirecionamento podem ser alteradas dinamicamente sem modificar programa.

3.4.4. Modos de Operação do XDP

Visando obter alto desempenho, programas eBPF carregados ao gancho XDP alteram o processamento de pacotes em nível de placa de rede, o que requer suporte explícito do driver associado. Alguns drivers para dispositivos de alta velocidade como `i40e`, `nfp`, `mlx*` e a família `ixgbe` já suportam essa funcionalidade, também chamada de *XDP nativo*. Nesses dispositivos programas eBPF são executados diretamente pelo driver, antes mesmo do pacote ser entregue ao sistema operacional. Uma lista atualizada dos drivers com suporte ao XDP nativo é mantida pelo projeto BCC [BCC 2019d].

No entanto, o *kernel* do Linux também oferece suporte para dispositivos que não

provêm essa funcionalidade em nível de driver através do modo *XDP genérico*. Nesse modo são criados buffers de soquete(*socket buffers*) especiais assim que os pacotes são recebidos pelo sistema operacional e a execução de programas XDP é feita pelo próprio *kernel*, emulando a execução do XDP nativo. Dessa forma até mesmo dispositivos que não têm suporte nativo ao XDP podem executar programas eBPF nesse gancho, ao custo de desempenho inferior por conta da alocação dos buffers [Miano et al. 2018].

A escolha entre esses dois modos de operação é feita automaticamente pelo sistema durante o carregamento do programa eBPF. Uma vez carregado, é possível verificar o modo de operação utilizando a ferramenta *iproute2* como mostrado na seção a seguir.

Ainda existe um terceiro modo de operação, chamado *XDP offload*. Como o nome sugere, o programa eBPF é descarregado para a placa de rede para ser executado em hardware, alcançando desempenho superior aos outros dois modos. Para habilitar esse modo, é necessário indicar explicitamente durante o carregamento do programa, além de um dispositivo capaz de executar eBPF em hardware. Atualmente as placas de rede inteligentes da empresa Netronome são os únicos dispositivos no mercado com essa capacidade.

3.4.5. Exemplo de gancho XDP e XDP offload

O exemplo a seguir requer um *kernel* atualizado a partir do Ubuntu 18.04 ou Fedora 28.

Para começar, escreve-se um programa C simples para processar pacotes de entrada e retornar XDP_DROP, o que impedirá que todos os pacotes recebidos cheguem ao host.

```c
#include <linux/bpf.h>
int main(){
    return XDP_DROP;
}
```

Depois de salvar o arquivo como xdp.c, o código pode ser compilado em um objeto eBPF usando o comando clang descrito a seguir. O compilador clang contém um *backend* eBPF permitindo que o código de montagem do eBPF seja gerado.

```
$ clang -target bpf -O2 -c xdp.c -o xdp.o
```

Depois que o programa é compilado, ele pode ser carregado usando a ferramenta *ip link* (altere [DEV] para o nome da interface relevante). No exemplo, o código não contém um rótulo de seção, portanto, o nome da seção ELF padrão .text é usado para o carregamento.

```
$ ip -force link set dev [DEV] xdpdrv obj xdp.o sec .text
```

Depois que o programa for carregado, o ip link mostrará que o programa eBPF está conectado à interface no gancho do XDP.

```
$ ip link show dev [DEV]
6: DEV: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc
mq state UP mode DEFAULT group default qlen 1000
    link/ether 00:15:4d:13:08:80 brd ff:ff:ff:ff:ff:ff
    prog/xdp id 27 tag f95672269956c10d jited
```

A palavra `xdp` na primeira linha da saída do comando indica que o programa está carregado no modo *XDP nativo*. Outros valores possíveis são `xdpgeneric` e `xdpoffload` para os modos *XDP genérico* e *XDP offload*, respectivamente. Para descarregar o programa XDP, um comando ip link semelhante pode ser usado, mas com o parâmetro off.

```
$ ip link set dev [DEV] xdpdrv off
```

3.4.5.1. Transferindo a computação para a placa de rede

No exemplo anterior, o eBPF foi executado no gancho XDP que está localizado dentro do driver e utiliza a CPU host x86 para o manuseio de pacotes. Se a interface de rede suportar, pode-se transferir a computação (descarregamento) do programa eBPF diretamente no processador da placa de rede. Atualmente, o único hardware com esse suporte é o Netronome Agilio CX SmartNIC.

Para descarregar o programa, o mesmo método pode ser utilizado, a única exceção é que *xdpoffload* é usado dentro do comando *ip link*.

```
$ ip -force link set dev [DEV] xdpoffload obj xdp.o sec .text
```

Para descarregar o programa, defina *xdpoffload* como desligado (off).

```
$ ip link set dev [DEV] xdpoffload off
```

3.4.6. Comparação entre XDP e DPDK

O XDP às vezes é comparado ao DPDK. O XDP oferece outra opção para usuários que desejam desempenho e ainda aproveitam a capacidade de programação do *kernel*. Algumas das funções que o XDP oferece incluem o seguinte:

- Remove a necessidade de código e licenciamento de terceiros
- Permite a opção de varredura ou interrupção
- Remove a necessidade de alocar páginas grandes
- Remove a necessidade de CPUs dedicadas, pois os usuários têm mais opções na estruturação de trabalho entre CPUs

- Remove a necessidade de injetar pacotes no *kernel* de um aplicativo de espaço do usuário de terceiros
- Remove a necessidade de definir um novo modelo de segurança para acessar o hardware de rede

3.4.7. Gancho *Traffic Control* (TC)

A pilha de rede do *kernel* do Linux possui uma camada para executar políticas de controle de tráfego, a camada de *Traffic Control* (TC). Nela o administrador de rede é capaz de configurar diferentes disciplinas para as diversas filas de pacotes do sistema operacional, assim como adicionar filtros para recusar ou modificar pacotes.

O TC apresenta um tipo de disciplina especial chamada `clsact`. Ela permite que as ações de processamento da fila sejam definidas por um programa eBPF. O pacote a ser processado é entregue para o programa eBPF configurado, no qual pode ser modificado, e o valor de retorno indica para o TC qual ação deve ser tomada pela fila. Os valores de retorno disponíveis estão definidos no arquivo `linux/pkt_cls.h`, sendo os mais comuns listados na tabela 3.3.

Tabela 3.3. Descrição do conjunto de ações do TC

Valor	Ação	Descrição
0	TC_ACT_OK	Dá prosseguimento ao pacote na fila do TC.
2	TC_ACT_SHOT	Descarta o pacote.
-1	TC_ACT_UNSPEC	Usa a ação padrão do TC.
3	TC_ACT_PIPE	Executa a próxima ação, se existir.
1	TC_ACT_RECLASSIFY	Reinicia a classificação desde o início.

O carregamento de programas no gancho TC é feito utilizando a ferramenta `tc`, disponível no pacote `iproute2`. O comando a seguir ilustra como criar a discplina `clsact` e carregar um código eBPF para processar os pacotes na interface `eth0`:

```
$ tc qdisc add dev eth0 clsact
$ tc filter add dev eth0 <direction> bpf da obj <ebpf-obj>
  ↪ sec <section>
```

O parâmetro `<direction>` indica a qual direção o programa deve ser associado, podendo ser `ingress` para RX ou `egress` para TX. `<ebpf-obj>` e `<section>` devem ser os nomes do arquivo que contém o código eBPF compilado e da seção correspondente ao programa que deseja ser adicionado, respectivamente.

Para verificar se há algum programa já carregado na interface `eth0`, basta utilizar o seguinte comando:

```
$ tc filter show dev eth0 <direction>
```

3.5. Programas eBPF

O eBPF suporta diferentes tipos de aplicações, por exemplo, medições de desempenho, filtragem e classificação de tráfego. Uma das principais vantagens do eBPF é fornecer um ambiente de programação flexível e seguro, onde programas podem ser escritos para diferentes contextos no *kernel* (*kernel probes*, eventos *perf*, dentre outros) [Maguire 2019].

Todo programa eBPF é verificado no momento em que é carregado para garantir que ele possa ser executado sem riscos dentro do *kernel*. Além disso, o eBPF suporta a compilação JIT de seu código de bytes para o conjunto de instruções nativo da máquina, o que torna os programas rápidos.

Programas eBPF possuem tipos que determinam os contextos em que eles podem executar. Eles utilizam estruturas de dados chamadas de mapas para armazenarem tipos de dados diferentes. Manipulação dos dados armazenados e realização de interações com o *kernel* são feitas através de funções especiais, chamadas de funções auxiliares.

3.5.1. Tipos de programas

O tipo de um programa eBPF define três pontos importantes: qual é a entrada do programa (o contexto), quais funções auxiliares ele pode utilizar e onde ele será carregado no *kernel*. Por exemplo, a entrada para um programa de filtragem de soquete é um pacote da rede, enquanto para um programa de rastreamento é um conjunto de valores dos registradores. De forma similar, o subconjunto das funções auxiliares que um programa de filtragem de soquete pode utilizar não é igual a um programa de rastreamento, embora possam existir funções em comum entre eles.

Os tipos de programas eBPF suportados são definidos através da *enum* `bpf_prog_type` em bpf.h, e podem crescer com novas versões do *kernel*. Na versão 5.0 do *kernel*, são disponibilizados um total de 23 tipos de programas diferentes. Alguns deles são:

- `BPF_PROG_TYPE_SOCKET_FILTER`: programa de filtragem de soquete;
- `BPF_PROG_TYPE_KPROBE`: programa para instrumentação em funções do *kernel* via *kprobe*;
- `BPF_PROG_TYPE_SCHED_CLS`: programa de classificação de controle de tráfego de rede;
- `BPF_PROG_TYPE_XDP`: programa XDP;
- `BPF_PROG_TYPE_PERF_EVENT`: programa para instrumentação de eventos *perf* de software e hardware;
- `BPF_PROG_TYPE_LWT_IN`, `BPF_PROG_TYPE_LWT_OUT` e `BPF_PROG_TYPE_LWT_XMIT`: programas para implementação de túneis leves (*lightweight tunnels*).
- `BPF_PROG_TYPE_SOCK_OPS`: programa que permite a obtenção de operações de soquetes (estabelecimento de conexão, tempo limite de retransmissão etc) e a definição de parâmetros de soquetes;

- `BPF_PROG_TYPE_SK_SKB`: programa que permite o acesso ao *skb* (*socket buffer*) e a detalhes de soquetes (endereço IP, porta etc) com o objetivo de realizar redirecionamento entre soquetes.

A lista dos tipos de programas suportados pode ser obtida do código fonte do Linux utilizando o seguinte comando:

```
$ git grep -W 'bpf_prog_type {' include/uapi/linux/bpf.h
```

3.5.2. Mapas

Mapas são estruturas de dados genéricas com a função de armazenar diferentes tipos de dados através de pares chave e valor. Tipos de dados são tratados como *blobs* binários e cabe ao usuário definir o tamanho das chaves e valores durante a criação do mapa.

Mapas eBPF podem ser criados de duas formas: por programas de espaço de usuário utilizando a chamada de sistema `bpf`, podendo ser manipulados através do descritor de arquivo retornado por ela em caso de sucesso, ou diretamente por programas eBPF carregados no *kernel*. No primeiro caso, a criação é feita com a utilização do comando `BPF_MAP_CREATE` (do *enum* `bpf_cmd`) e de uma *union* de configuração `bpf_attr` como parâmetros da chamada de sistema

```
bpf(BPF_MAP_CREATE, &bpf_attr, sizeof(bpf_attr)).
```

A *union* de configuração requer a definição dos seguintes atributos:

1. *map_type*: tipo do mapa a ser criado;
2. *key_size*: tamanho das chaves em bytes;
3. *value_size*: tamanho dos valores em bytes;
4. *max_entries*: quantidade máxima de entradas no mapa.

Um processo no espaço de usuário pode criar múltiplos mapas. Mapas podem ser acessados paralelamente por outros programas eBPF. Isso permite o compartilhamento de dados entre aplicações eBPF dentro *kernel*, e entre aplicações no *kernel* e espaço de usuário. Para destruir um mapa, o descritor de arquivo associado a ele deve ser fechado.

Cada programa eBPF deve declarar os mapas utilizados como variáveis globais na seção *maps* utilizando a estrutura `bpf_map_def` definida pela `libbpf` (§3.6.1) . O exemplo a seguir declara um mapa `rxcnt` do tipo `BPF_PROG_TYPE_PERCPU_ARRAY`:

```
struct bpf_map_def SEC("maps") rxcnt = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(uint32_t),
    .value_size = sizeof(long),
    .max_entries = 256,
};
```

3.5.3. Tipos de mapas

Existem diferentes tipos de mapas disponíveis para programas eBPF definidos no *enum* `bpf_map_type`. Cada tipo oferece um determinado comportamento, alguns deles são utilizados de forma genérica, outros para casos de uso específicos.

Exemplos de código de mapas eBPF são providas pelo *kernel* do Linux. A versão 5.0 do *kernel* lista um total de 23 mapas diferentes. Alguns deles são descritos:

- `BPF_MAP_TYPE_HASH`: tabela hash genérica.
- `BPF_MAP_TYPE_ARRAY`: vetor otimizado para pesquisas rápidas. É frequentemente utilizado por aplicações que utilizam contadores.
- `BPF_MAP_TYPE_PROG_ARRAY`: vetor de descritores de arquivos correspondentes a programas eBPF. Sua utilização permite, por exemplo, a chamada de subprogramas para lidar com situações específicas.
- `BPF_MAP_TYPE_PERCPU_HASH`: mapa similar ao `BPF_MAP_TYPE_HASH`. Permite a criação de uma tabela hash para cada núcleo do processador.
- `BPF_MAP_TYPE_PERCPU_ARRAY`: mapa similar ao `BPF_MAP_TYPE_ARRAY`. Permite a criação de um vetor para cada núcleo do processador.
- `BPF_MAP_TYPE_PERF_EVENT_ARRAY`: mapa para o armazenamento e leitura de contadores de eventos da ferramenta *perf*.
- `BPF_MAP_TYPE_LRU_HASH`: tabela hash que mantém somente os elementos utilizados pela última vez recentemente (LRU). Assim, quando a tabela se encontra cheia, os elementos utilizados pela última vez a mais tempo são removidos primeiramente.
- `BPF_MAP_TYPE_LRU_PERCPU_HASH`: versão que utiliza tabelas hash LRU do mapa `BPF_MAP_TYPE_PERCPU_HASH`.
- `BPF_MAP_TYPE_LPM_TRIE`: uma *trie* para o casamento do prefixo mais longo.
- `BPF_MAP_TYPE_STACK_TRACE`: mapa para armazenamento de *traces* de pilhas.
- `BPF_MAP_TYPE_ARRAY_OF_MAPS`: vetor para o armazenamento de mapas eBPF.
- `BPF_MAP_TYPE_HASH_OF_MAPS`: tabela hash para armazenamento de mapas eBPF.
- `BPF_MAP_TYPE_DEVMAP`: mapa para armazenamento e leitura de referências a dispositivos de rede.
- `BPF_MAP_TYPE_SOCKMAP`: mapa para armazenamento de soquetes. Pode ser utilizado para implementar redirecionamento de soquetes, por exemplo.
- `BPF_MAP_TYPE_QUEUE`: mapa com comportamento similar ao de uma fila.

- `BPF_MAP_TYPE_STACK`: mapa com comportamento similar ao de uma pilha.

A lista de todos os tipos de mapas suportados pode ser obtida diretamente do código fonte do Linux a partir do seguinte comando:

```
$ git grep -W 'bpf_map_type {' include/uapi/linux/bpf.h
```

3.5.4. Funções auxiliares

O eBPF se diferencia do BPF "clássico"(cBPF) em muitos aspectos. Um deles é a habilidade de permitir que programas façam a chamada de funções especiais denominadas funções auxiliares. Funções auxiliares permitem que programas eBPF sejam capazes de interagir com o sistema e com o contexto ao qual essas funções trabalham. Exemplos de tarefas realizadas por funções auxiliares incluem interação com mapas eBPF, manipulação de pacotes da rede e impressão de mensagens de depuração.

As funções auxiliares disponíveis para os programas eBPF são restritas as funções definidas no *kernel*. Como existem diferentes tipos de programa eBPF e eles não executam em um mesmo contexto, cada tipo de programa pode chamar somente um subconjunto dessas funções. Por exemplo, algumas funções auxiliares só estão disponíveis para programas XDP. O projeto BCC oferece uma documentação atualizada das funções auxiliares disponíveis para cada tipo de programa eBPF [BCC 2019b].

Novas funções auxiliares podem ser adicionadas somente através de extensões do *kernel*. Isso significa que nenhuma função auxiliar pode ser estendida ou adicionada através da adição de módulos no *kernel*. Inserir uma nova função auxiliar requer o seguimento de uma assinatura convencionada pelo eBPF. Essa assinatura é compartilhada por todas as funções auxiliares, limitando a quantidade de argumentos para no máximo cinco. Uma assinatura é definida como:

```
u64 fn(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)
```

Toda nova função auxiliar adicionada é compilada em tempo de execução de forma transparente e eficiente. Isso significa que o compilador JIT necessita somente emitir uma instrução de chamada, pois o mapeamento dos registradores eBPF é feito de forma a corresponder com a convenção de chamada da arquitetura subjacente. Programas eBPF chamam diretamente as funções auxiliares compiladas e por isso elas não introduzem nenhum *overhead*, oferecendo um bom desempenho.

O número de funções auxiliares atualmente disponíveis é amplo e cresce ao longo do tempo. A versão 5.0 do *kernel* disponibiliza um total de 92 funções diferentes. Algumas delas são descritas:

- `bpf_map_lookup_elem`, `bpf_map_update_elem` e `bpf_map_delete_elem`: funções utilizadas, respectivamente, para pesquisar, atualizar e remover uma entrada associada a uma chave no mapa.
- `bpf_get_prandom_u32`: retorna um valor de 32 bits pseudo-aleatório.

- `bpf_tail_call`: função utilizada para pular para outro programa eBPF.
- `bpf_l3_csum_replace` e `bpf_l4_csum_replace`: funções utilizadas para recalcular o *checksum* das camadas 3 (ex. IP) e 4 (ex. TCP ou UDP) de pacotes.
- `bpf_ktime_get_ns`: retorna o tempo decorrido desde o *boot* do sistema, em nanosegundos.
- `bpf_clone_redirect` e `bpf_redirect`: funções utilizadas para redirecionar pacotes para um dispositivo de rede. A diferença entre elas é que a primeira clona o pacote, enquanto a segunda não (o que garante melhor desempenho).
- `bpf_skb_vlan_push` e `bpf_skb_vlan_pop`: funções para a adição (*push*) e remoção (*pop*) de um cabeçalho VLAN de um pacote.
- `bpf_getsockopt` e `bpf_setsockopt`: funções que emulam as respectivas chamadas de `getsockopt()` e `setsockopt()` para soquetes.
- `get_local_storage`: retorna um ponteiro para uma área de armazenamento local. Dependendo do tipo do programa eBPF, essa área de armazenamento pode ser compartilhada entre múltiplas instâncias do programa rodando paralelamente.

A declaração de todas as funções auxiliares é feita no arquivo `bpf_helpers.h`, localizado no diretório `tools/testing/selftests/bpf`. Outras operações comuns durante o processamento de pacotes como a conversão de valores binários entre *little* e *big endian* estão disponíveis no arquivo `bpf_endian.h`, localizado no mesmo diretório. Este arquivo oferece versões de funções conhecidas como `ntohs` e `htons`, na forma de funções auxiliares (`bpf_ntohs` e `bpf_htons`, respectivamente), assim como outras funções similares.

Além de incluir essas bibliotecas no código fonte do programa eBPF, é necessário passar o caminho do diretório onde elas estão localizadas para que o compilador saiba como importar os arquivos. Isso pode ser feito utilizando a *flag* `-I` do *clang*. Caso não deseje trabalhar diretamente com o código do *kernel* (ex: evitar dependências externas), é possível usar cópias locais destes dois arquivos. A geração do código de máquina das funções auxiliares é feita durante a compilação do próprio *kernel* e as chamadas em tempo de execução são feitas internamente pelo sistema operacional. Por conta disso, para gerar o executável de um programa eBPF o compilador só necessita saber a assinatura das funções auxiliares, possibilitando o uso deste artifício.

3.5.5. Exemplos

O código fonte do Linux contém diversos exemplos de programas eBPF, localizados em dois diretórios distintos: `samples/bpf` e `tools/testing/selftests/bpf`. Ambos contém programas que demonstram o uso de diversas funcionalidades e ganchos da pilha do *kernel*, com novos programas sendo adicionados a cada nova versão do *kernel*.

A maior parte dos exemplos do primeiro diretório está dividida em dois arquivos separados, um com código em espaço de usuário para carregar o programa (arquivos

terminados em *user.c*) e o outro com a implementação do programa eBPF a ser carregado no *kernel* (arquivos terminados em *kern.c*).

Já os exemplos do segundo diretório são utilizados como base para a execução de testes funcionais durante o desenvolvimento do arcabouço eBPF. Por conta disso, a tendência da comunidade de desenvolvedores é adicionar os programas mais recentes ao segundo diretório, no formato de testes de unidade. Em geral estes exemplos utilizam as últimas novidades, representando melhor o estado atual e as ferramentas disponíveis para o desenvolvimento de programas eBPF.

A seguir é apresentado o exemplo `xdp1`, presente no diretório `samples/bpf`. O arquivo `xdp1_kern.c` contém um programa eBPF que processa pacotes no gancho XDP e armazena a contagem do número de pacotes recebidos por protocolo de camada de rede em um mapa.

```
8  #include <uapi/linux/bpf.h>
9  #include <linux/in.h>
10 #include <linux/if_ether.h>
11 #include <linux/if_packet.h>
12 #include <linux/if_vlan.h>
13 #include <linux/ip.h>
14 #include <linux/ipv6.h>
15 #include "bpf_helpers.h"
16
17 struct bpf_map_def SEC("maps") rxcnt = {
18     .type = BPF_MAP_TYPE_PERCPU_ARRAY,
19     .key_size = sizeof(u32),
20     .value_size = sizeof(long),
21     .max_entries = 256,
22 };
23
24 static int parse_ipv4(void *data, u64 nh_off, void *data_end) {
25     struct iphdr *iph = data + nh_off;
26
27     if (iph + 1 > data_end)
28         return 0;
29     return iph->protocol;
30 }
31
32 static int parse_ipv6(void *data, u64 nh_off, void *data_end) {
33     struct ipv6hdr *ip6h = data + nh_off;
34
35     if (ip6h + 1 > data_end)
36         return 0;
37     return ip6h->nexthdr;
38 }
39
40 SEC("xdp1")
41 int xdp_prog1(struct xdp_md *ctx) {
42     void *data_end = (void *)(long)ctx->data_end;
43     void *data = (void *)(long)ctx->data;
44     struct ethhdr *eth = data;
45     int rc = XDP_DROP;
46     long *value;
```

```
47      u16 h_proto;
48      u64 nh_off;
49      u32 ipproto;
50
51      nh_off = sizeof(*eth);
52      if (data + nh_off > data_end)
53          return rc;
54
55      h_proto = eth->h_proto;
56
57      if (h_proto == htons(ETH_P_8021Q)  h_proto == htons(ETH_P_8021AD))
          ↪  {
58          struct vlan_hdr *vhdr;
59
60          vhdr = data + nh_off;
61          nh_off += sizeof(struct vlan_hdr);
62          if (data + nh_off > data_end)
63              return rc;
64          h_proto = vhdr->h_vlan_encapsulated_proto;
65      }
66      if (h_proto == htons(ETH_P_8021Q)  h_proto == htons(ETH_P_8021AD))
          ↪  {
67          struct vlan_hdr *vhdr;
68
69          vhdr = data + nh_off;
70          nh_off += sizeof(struct vlan_hdr);
71          if (data + nh_off > data_end)
72              return rc;
73          h_proto = vhdr->h_vlan_encapsulated_proto;
74      }
75
76      if (h_proto == htons(ETH_P_IP))
77          ipproto = parse_ipv4(data, nh_off, data_end);
78      else if (h_proto == htons(ETH_P_IPV6))
79          ipproto = parse_ipv6(data, nh_off, data_end);
80      else
81          ipproto = 0;
82
83      value = bpf_map_lookup_elem(&rxcnt, &ipproto);
84      if (value)
85          *value += 1;
86
87      return rc;
88  }
89  char _license[] SEC("license") = "GPL";
```

Múltiplos programas eBPF podem residir no mesmo arquivo `.c`, sendo separados em diferentes seções no arquivo ELF gerado pelo compilador. O rótulo de seção acima da função correspondente (Linha 40) indica para o compilador o nome da seção ELF que conterá o programa no arquivo objeto gerado. Essa informação é necessária durante o carregamento do código no *kernel* para que o sistema saiba qual seção ELF deve ser carregada. Caso o arquivo contenha apenas um programa, o rótulo pode ser omitido, e o programa será carregado na seção padrão `.text`. Rótulos de seção também são utilizados durante a declaração de mapas (Linha 17) e da licença do programa (Linha 89),

sendo ambos obrigatórios e com valores fixos. A seção de licença é usada pelo verificador para determinar quais funções auxiliares estarão disponíveis para o usuário, pois algumas funções só podem ser usadas por programas que declaram a mesma licença.

O programa em questão foi desenvolvido para ser carregado no gancho XDP, portanto o parâmetro de entrada da função deve ser do tipo `struct xdp_md`, representando o contexto passado por esse gancho para o programa eBPF. Os bytes do pacote sendo processado são delimitados pelos ponteiros `data` e `data_end`, que devem ser usados durante todo o programa para efetuar acessos ao pacote. As conversões de tipos para esses dois valores são padrão, de forma que as Linhas 42 e 43 devem ser utilizadas no início de todo programa eBPF.

Para garantir a integridade e a segurança do *kernel*, o verificador do sistema eBPF (§3.3.3) não permite acessos de memória além das variáveis locais e dos limites do pacote indicados pelo contexto passado. Para acessar quaisquer bytes do pacote, sempre é necessário fazer uma verificação de limites, como demonstrado nas Linhas 27, 35, 52, 62 e 71. Porém, cada byte só precisa ser verificado uma única vez, a não ser que funções auxiliares que modificam o espaço de armazenamento do pacote sejam usadas, como `bpf_xdp_adjust_head`. Nesse caso toda a checagem deve ser refeita após a chamada dessa função. Durante a análise do programa o verificador garante que todos os acessos de memória feitos ao pacote são em endereços verificados dessa forma. Caso esse tipo de checagem não seja feita dentro do programa eBPF, ele é rejeitado pelo verificador durante o carregamento no *kernel*.

Após determinar o tipo do protocolo de camada 3 do pacote, o programa atualiza o contador para o protocolo correspondente utilizando a função auxiliar de consulta (Linha 83). Essa função retorna um ponteiro para o valor atual armazenado no mapa, caso ele exista, ou `NULL` caso contrário. Esse endereço pode ser usado para alterar o dado armazenado de forma direta, sem a necessidade de uma operação de atualização do mapa. Por fim, o programa retorna a ação que deve ser tomada pelo gancho XDP para o pacote atual, que nesse caso é sempre `XDP_DROP`, indicando que o pacote deve ser descartado.

As estatísticas armazenadas no mapa `rxcnt` são então consultadas por uma aplicação em espaço de usuário, implementada pelo arquivo `xdp1_user.c`. Por questões de espaço destacamos abaixo apenas alguns trechos deste programa.

O programa inclui as bibliotecas `libbpf.h` (§3.6.1) e `bpf.h` para permitir interação com o sistema eBPF:

```
21 #include "bpf/bpf.h"
22 #include "bpf/libbpf.h"
```

A informações do programa a ser carregado são passadas por meio da estrutura `bpf_prog_load_attr`, incluindo o tipo de programa, o arquivo objeto contendo o programa e o identificador da interface ao qual ele deve ser associado.

```
73 struct bpf_prog_load_attr prog_load_attr = {
74     .prog_type        = BPF_PROG_TYPE_XDP,
75 };
```

```
112 snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);
113 prog_load_attr.file = filename;
```

Essa estrutura então é utilizada para carregar o programa no gancho XDP. Em caso de sucesso, após a chamada as variáveis `obj` e `prog_fd` contém as informações detalhadas do código já carregado e o seu descritor de arquivo, respectivamente. O descritor é utilizado para identificar o programa dentre os demais atualmente carregados no kernel, sendo necessário para interações futuras com esse programa.

```
115 if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
116     return 1;
```

Após o carregamento do programa eBPF no *kernel*, obtém-se a referência para o mapa `rxcnt`. A função `bpf_map__next` retorna um iterador para a lista de mapas declarados no programa. Como nesse caso há apenas um mapa declarado, o valor desse iterador por ser utilizado para se obter o descritor de arquivo referente a ele. A *libbpf.h* também oferece outras funções para se obter descritores de mapa por nome ou por índice na lista de mapas.

```
118 map = bpf_map__next(NULL, obj);
119 if (!map) {
120     printf("finding a map in obj file failed\n");
121     return 1;
122 }
123 map_fd = bpf_map__fd(map);
```

Finalmente, o programa eBPF está pronto para ser carregado no gancho XDP e a aplicação em espaço de usuário pode entrar no laço infinito da função `poll_stats`.

```
130 if (bpf_set_link_xdp_fd(ifindex, prog_fd, xdp_flags) < 0) {
131     printf("link set xdp fd failed\n");
132     return 1;
133 }
134
135 poll_stats(map_fd, 2);
```

A função `poll_stats`, utilizando o descritor de arquivo correspondente ao mapa `rxcnt`, executa consultas periódicas ao mapa e lista todas as entradas existentes, juntamente com as estatísticas calculadas até o momento.

```
35 static void poll_stats(int map_fd, int interval)
36 {
37     unsigned int nr_cpus = bpf_num_possible_cpus();
38     __u64 values[nr_cpus], prev[UINT8_MAX] = { 0 };
39     int i;
40
41     while (1) {
42         __u32 key = UINT32_MAX;
```

```
43
44          sleep(interval);
45
46          while (bpf_map_get_next_key(map_fd, &key, &key) != -1) {
47              __u64 sum = 0;
48
49              assert(bpf_map_lookup_elem(map_fd, &key, values) == 0);
50              for (i = 0; i < nr_cpus; i++)
51                  sum += values[i];
52              if (sum > prev[key])
53                  printf("proto %u: %10llu pkt/s\n",
54                         key, (sum - prev[key]) / interval);
55              prev[key] = sum;
56          }
57      }
58  }
```

Apesar de simples, esse exemplo demonstra algumas questões básicas que devem ser levadas em conta durante o desenvolvimento de programas eBPF, além de mostrar como pode ser feita a interação com programas em espaço de usuário.

3.6. Ferramentas

Esta seção apresentada algumas ferramentas que podem ser de grande utilidade para o desenvolvimento e depuração de programas eBPF.

3.6.1. libbpf

O *kernel* fornece a biblioteca *libbpf* [libbpf 2018], localizada em `tools/lib/bpf`. Ela inclui funções auxiliares para carregar programas e criar e manipular objetos eBPF em espaço de usuário. Para incluir essa biblioteca é necessário informar ao compilador o caminho para o diretório:

```
clang -I <linux-code-dir>/tools/lib ... myprog.c
```

Também é necessário incluir a biblioteca no código C:

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

Diversos exemplos disponíveis no diretório `tools/testing/selftests/bpf` demonstram casos de uso da biblioteca, podendo servir como um bom ponto de partida.

3.6.2. iproute2

O *iproute2* é uma coleção de utilitários do espaço de usuário para controlar e monitorar vários aspectos da rede no *kernel* do Linux, incluindo roteamento, interfaces de rede, túneis, controle de tráfego e drivers de dispositivos relacionados à rede. Dentre os produtos mais conhecidos estão as ferramentas `ip` e `tc`. Ambas oferecem maneiras alternativas de carregar códigos eBPF no *kernel*, sem a necessidade de um programa em espaço de

usuário fazendo uso da biblioteca *libbpf*. Com o comando `ip`, por exemplo, é possível carregar códigos no gancho XDP, como mostrado na seção 3.4.5, assim como em níveis mais altos, como no roteamento em camada 3:

```
$ ip route add 192.168.0.0/24 encap bpf headroom 22 xmit
  ↪ obj <bpf-prog> section <section> dev eth0
```

O comando acima adiciona uma regra de roteamento para a subrede 192.168.0.0/24 que executa um programa eBPF localizado na seção `<section>` do arquivo objeto `<bpf-prog>` e encapsula os pacotes com 22 bytes iniciais.

Além disso, o *iproute2* tem a sua própria interface de interação com o sistema eBPF, oferecendo funcionalidades adicionais como a possibilidade de especificar o escopo de alocação de mapas eBPF. Para isso, é necessário declarar um mapa utilizando uma estrutura alternativa (`bpf_elf_map`), definida em `iproute2/include/bpf_elf.h`, e incluir essa estrutura no código C:

```
#include <linux/bpf.h>
#include <iproute2/bpf_elf.h>

struct bpf_elf_map SEC("maps") src_mac = {
    .type = BPF_MAP_TYPE_HASH,
    .size_key = 1,
    .size_value = 6,
    .max_elem = 1,
    .pinning = PIN_GLOBAL_NS,
};
```

Essa estrutura é similar a `bpf_map_def` da *libbpf* porém apresenta campos extras, como o campo `pinning`, utilizado para definir o escopo do mapa, podendo assumir três valores distintos: `PIN_OBJECT_NS`, `PIN_GLOBAL_NS` ou `PIN_NONE`.

Mapas criados com `PIN_OBJECT_NS` têm escopo local, sendo exclusivos do programa que os declarou. Como consequência, mapas com a mesma declaração podem co-existir em programas distintos. Nesse caso, um diretório específico será criado no pseudo-sistema de arquivos BPF para armazenar os nós correspondentes a esses mapas. Caso o valor `PIN_OBJECT_GLOBAL` seja usado, o mapa é criado com um escopo global, habilitando o seu compartilhamento por múltiplos programas. Esse mapa receberá uma entrada no diretório `globals` no pseudo-sistema de arquivos. Por último, o valor `PIN_NONE` indica que o mapa não deve ser fixado no pseudo-sistema de arquivos, impossibilitando a interação com outras aplicações em espaço de usuário.

3.6.3. bpftool

O *bpftool* [bpftool 2018] é uma ferramenta no espaço de usuário essencial para depuração e inserção de programas e mapas BPF. Faz parte da árvore do *kernel* e está disponível em *`tools/bpf/bpftool/`*. Ele pode ser usado para coletar informações sobre programas e mapas do eBPF. Por exemplo, pode-se listar programas carregados:

```
# bpftool prog show
27: xdp tag b722a8b5b9e9be25 dev ens4np0
loaded_at Jun 12/13:20 uid 0
xlated 112B jited 392B memlock 4096B map_ids 31
```

Pode-se imprimir as instruções para este programa com o comando:

```
# bpftool prog dump xlated id 27
0: (b7) r1 = 0
1: (63) *(u32 *)(r10 -4) = r1
2: (bf) r2 = r10
3: (07) r2 += -4
4: (18) r1 = map[id:31]
6: (85) call 0x0#1725914768
7: (b7) r1 = 1
8: (15) if r0 == 0x0 goto pc+3
9: (b7) r1 = 1
10: (c3) lock *(u32 *)(r0 +0) += r1
11: (b7) r1 = 2
12: (bf) r0 = r1
13: (95) exit
```

Além disso, pode-se listar e imprimir o conteúdo de mapas também:

```
# bpftool map
1234: array name ch_rings flags 0x0
key 4B value 4B max_entries 7860 memlock 65536B
# bpftool map dump id 1234
key: 00 00 00 00 value: 00 00 00 00
key: 01 00 00 00 value: 00 00 00 00
key: 02 00 00 00 value: 00 00 00 00
key: 03 00 00 00 value: 00 00 00 00
[...]
Found 7860 elements
```

Também é possível executar algumas operações de gerenciamento, incluindo carregar programas, realizando pesquisas ou atualizações de valores de mapa. Um exemplo para o último item:

```
# bpftool map update id 1234 key 0x01 0x00 0x00 0x00 value 0x12 0x34
  ↪  0x56 0x67
```

3.6.4. llvm-objdump

A ferramenta `llvm-objdump` (versão 4.0 ou superior) pode ser usada para descarregar o código de byte compilado em um formato legível para seres humanos, antes que o usuário tente injetá-lo no *kernel*. Além disso, ela é útil para inspecionar as seções ELF do arquivo eBPF compilado.

```
$ llvm-objdump -S sample_ret0.o
sample_ret0.o: file format ELF64-BPF
```

```
Disassembly of section .text:
func:
; {
0: b7 00 00 00 00 00 00 00 r0 = 0
; return 0;
1: 95 00 00 00 00 00 00 00 exit
```

3.6.5. BPF Compiler Collection (BCC)

O projeto de código aberto *BPF Compiler Collection* (BCC) [BCC 2019a] tem como um de seus objetivos principais facilitar o desenvolvimento de programas baseados em eBPF. Ele oferece um conjunto de *frontends* de compiladores que podem ser usados para interagir com o sistema eBPF utilizando linguagens de alto nível como Python, além de compilar código eBPF diretamente para a linguagem P4 [BCC 2019c].

Além disso, o projeto conta com uma série de ferramentas de exemplo, construídas sobre o BCC, capazes de desempenhar diversas tarefas no sistema operacional. Essas ferramentas podem realizar tarefas como analisar o número de chamadas de sistema executadas por uma aplicação e o tempo decorrido durante uma leitura do disco. Por serem baseadas em programas eBPF são utilizadas por grandes empresas para analisar sistemas de produção reais com baixa sobrecarga adicional.

3.7. Plataformas

Atualmente existem algumas plataformas distintas que utilizam o conjunto de instruções eBPF para adicionar programabilidade em diferentes contextos. Nesta seção são apresentadas as principais delas, divididas em software ou hardware.

3.7.1. Software

A seguir são descritas as plataformas capazes de processar programas eBPF em software, baseadas na implementação original do *kernel* do Linux.

3.7.1.1. *Kernel* do Linux

Como discutido anteriormente, o *kernel* do Linux foi o berço dessa tecnologia e é a plataforma mais popular e com maior desenvolvimento, sendo o foco principal deste texto. Atualmente as aplicações do eBPF no *kernel* já não estão restritas somente à pilha de rede, mas também englobam diversas atividades de instrumentação e monitoramento do sistema operacional. Logo, programas eBPF se tornaram uma importante ferramenta de introspecção utilizada, por exemplo, por projetos de código aberto como IOVisor [IOvisor 2019].

3.7.1.2. Userspace BPF

O projeto de código aberto *ubpf* [ubpf 2019] é uma adaptação da máquina virtual eBPF presente no Linux para o espaço de usuário. Utilizando o interpretador ou o compilador JIT fornecidos por ele é possível embutir uma máquina eBPF em outras aplicações de espaço de usuário, tirando proveito da flexibilidade do conjunto de instruções eBPF.

Muito similar ao *ubpf*, o projeto *rbpf* oferece a mesma funcionalidade porém utilizando a linguagem Rust [Monnet 2019].

Entretanto, diferentemente do sistema eBPF no *kernel*, o *ubpf* não apresenta suporte a mapas e não tem funções auxiliares já implementadas. Apesar disso, ele pode ser estendido para suportar essas operações, como foi feito por Jouet e Pezaros, que o utilizaram como base para o desenvolvimento do comutador BPFabric, discutido a seguir.

3.7.1.3. BPFabric

A arquitetura OpenFlow [McKeown et al. 2008], reconhecida como sendo a implementação *de facto* de SDN, promove uma separação dos planos de controle e de dados das redes e provê uma forma de controlar o *pipeline* de casamento (*match*) do switch utilizando um conjunto limitado de ações e campos. Ela foi criada como um padrão aberto e independente de fornecedor, o que permite que ela seja utilizada em switches em hardware ou software de diferentes fornecedores. Ela apresenta, no entanto, limitações relacionadas ao suporte de novos protocolos e novas ações de casamento. Novas versões do OpenFlow necessitam ser desenvolvidas sempre que inclusões de suporte a casamentos de novos campos ou de novas ações são desejadas.

Para lidar com as limitações do OpenFlow, em [Jouet and Pezaros 2017a] tem-se a proposta de uma nova arquitetura SDN chamada de BPFabric. Essa arquitetura é independente de plataforma, protocolo e linguagem e permite que o plano de dados seja programado a fim de que novas funções possam ser adicionadas ao switch. Tal independência é atingida através da adoção do eBPF como o conjunto de instruções para as funções do plano de dados compiladas.

Como o eBPF não possui conhecimento sobre nenhum protocolo de rede e nem sobre estruturas de pacotes, ele permite que diferentes protocolos possam ser analisados e que um conjunto de tabelas seja utilizado para manter estados e realizar operações de casamento e ação. Desta forma, novas funções podem ser utilizadas para dar suporte a novas funcionalidades e para prover serviços como telemetria, coleta de estatísticas e rastreamento de pacotes.

Para facilitar o desenvolvimento de novas funções, uma linguagem de alto nível restrita é utilizada definir o comportamento do plano de dados. Um exemplo de tal linguagem pode ser um subconjunto restrito de C, onde chamadas de sistemas e algumas bibliotecas externas são excluídas. O código produzido com a linguagem de alto nível é compilado para um arquivo eBFP de execução e ligação (ELF) e deve ser posteriormente carregado no switch pelo controlador. O código compilado define um novo comportamento do switch e reestrutura completamente o seu pipeline de casamento.

Assim como em outras arquiteturas SDN, no BPFabric o switch também é pensado para ser "burro". Nesse sentido, ele não contém nenhuma lógica interna e seu comportamento precisa explicitamente definido pelo controlador. A figura 3.6 mostra uma visão geral da arquitetura do switch, dividida entre os planos de controle e de dados.

O plano de controle foi pensado para ser simples, abrigando apenas um agente que é responsável por intermediar operações entre o controlador e o plano de dados através

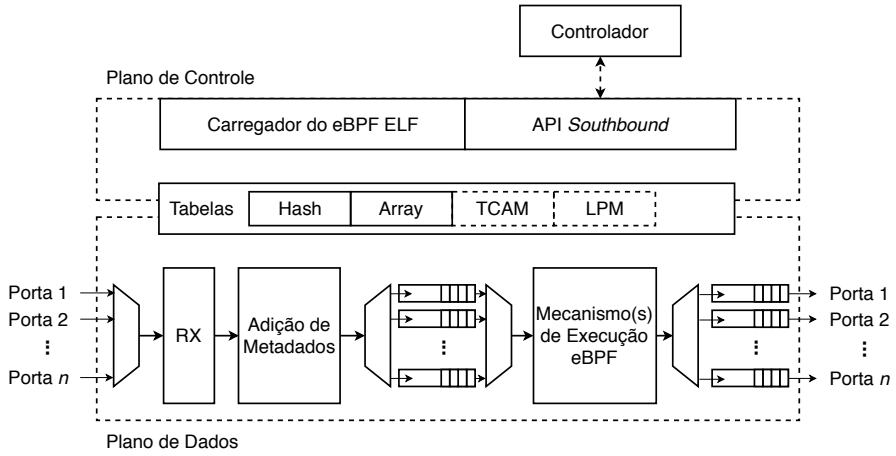


Figura 3.6. Visão simplificada da arquitetura do switch definida no BPFabric.

da API *Southbound*. Entre as operações realizadas pelo agente estão: (i) alterações no comportamento do switch, (ii) recebimento de pacotes e notificações de eventos, e (iii) leituras e atualizações de entradas das tabelas. Quando recebe o código compilado do controlador, o agente faz o uso do Carregador de eBPF ELF para modificar o pipeline do switch. O Carregador é responsável por primeiramente verificar a validade, segurança e o desempenho do programa eBPF a ser carregado. Em caso de sucesso, ele deve realizar a alocação das tabelas eBPF necessárias e converter o código de byte recebido em um formato específico para o switch. Isso faz dele um componente crítico do plano de controle e específico para cada dispositivo.

O plano de dados determina o fluxo descrito a seguir. Os pacotes recebidos pelas interfaces de entrada do switch são armazenados em filas de recebimento com metadados de *timestamp* e informações providas pela camada de enlace sobre eles. Quando um dos mecanismos de execução eBPF se torna disponível, um pacote e seus metadados são retirados de uma das filas de recebimento e então passam pelo pipeline de processamento. É neste estágio que as funções carregadas pelo controlador são executadas. Ao final do pipeline, tem-se o retorno da decisão de encaminhamento do pacote, podendo ela ser o envio dele para o controlador, para alguma porta de saída, para todas as portas de saída (*flood*) ou para nenhuma delas (*drop*).

3.7.2. Hardware

Além das plataformas em software, têm surgido também dispositivos em hardware capazes de executar códigos eBPF. Nesses casos o conjunto de instruções é utilizado para prover programabilidade aos dispositivos e também oferecer uma plataforma alternativa para execução de programas eBPF com maior desempenho.

3.7.2.1. Placa de interface de rede inteligente (*SmartNIC*)

Placas de interface de rede inteligentes (*SmartNICs*) são placas de rede que, além de prover conectividade, permitem que o processamento de tráfego de rede, que seria nor-

malmente feito pela CPU, seja implementado no próprio dispositivo. Elas geralmente proveem funcionalidades de programação e grandes quantidades de memória. Um exemplo de *SmartNIC* é a placa Netronome Agilio CX 2x10GbE [Beckett et al. 2018], que oferece 2 portas 10 GbE emparelhadas com um processador ARM11, 2 GB de memória RAM DDR3 e mais de 100 núcleos de processamento dedicado.

Devido à grande capacidade de processamento das *SmartNICs* atuais, elas têm se tornado um grande atrativo para o eBPF. O descarregamento do eBPF em *hardware* têm a capacidade de propiciar diversas vantagens e ganhos em desempenho. A redução da latência é uma das vantagens possíveis, visto que pacotes não precisam deixar a placa para serem processados. Outra vantagem é a habilidade de carregar programas *on-the-fly* que o eBPF propicia, permitindo a troca dinâmica de programas em um *data center* operante sem a necessidade de reinicialização dos sistemas. Além disso, a programação de *SmartNICs* através do eBPF facilita a implementação de recursos como limitadores de fluxos e filtragem de pacotes, o que permite o desenvolvimento de aplicações eficientes para a mitigação de ataques de negação de serviço (*Denial of Service* – DoS), balanceamento de cargas, comutação de pacotes, dentre outras.

3.7.2.2. Roteador eBPF em Hardware

Processamento de pacotes sobre hardwares com plano de dados programável tornaram-se um campo de pesquisa ativo em redes de computadores que ganhou atenção da indústria e academia da área [Dargahi et al. 2017]. Hardwares com essa característica possibilitam o processamento de pacotes de modo flexível e expressivo sem conhecer comandos e especificações de baixo nível do plano de dados via espaço do usuário. Usuários ao utilizar esse tipo de hardware podem analisar, processar e encaminhar pacotes utilizando tabelas de casamento e ações de modo dinâmico. Atualmente, hardwares como ASICs, CPUs x86, GPUs e FPGAs estão sendo utilizados no processamento de pacotes de redes *Gigabit Ethernet* [Sharma et al. 2017].

FPGAs comparado a outras plataformas de hardware tornaram-se mais atrativos para área de redes de computadores por combinar alto poder de processamento com flexibilidade. Empresas como Microsoft, Baidu e Amazon já inseriram FPGAs em seus provedores de nuvem para acelerar o processamento de pacotes de suas redes. FPGA é um dispositivo composto por blocos lógicos programáveis, memórias e dispositivos de entrada e saída. A programação neste equipamento ocorre utilizando linguagens de descrição de hardware (LDHs), por exemplo, Verilog ou VHDL [Wang et al. 2017].

Aplicações de rede são implementadas sobre FPGAs sintetizando programas descritos usando LDHs, e carregando o arquivo *Bitstream* gerado após a síntese. Para carregar um arquivo *Bitstream* o usuário pára o funcionamento do equipamento. Além disso, uma síntese pode levar horas e cada modificação realizada no circuito acarreta uma nova síntese. Usuários gastam horas simulando e reescrevendo circuitos que não funcionam após síntese. Todas essas dificuldades em conjunto tornam o processo de desenvolvimento de aplicações de rede uma tarefa complexa e desafiadora que demanda muito esforço e conhecimento do hardware por parte do usuário [Zilberman et al. 2015].

Para lidar com as limitações do OpenFlow e FPGAs, [Pacífico et al. 2018] propôs

uma arquitetura de roteador eBPF em hardware que processa pacotes independente de protocolo, e utiliza instruções eBPF geradas a partir de programas C ou P4 criados pelo usuário para definir como os pacotes serão processados no plano de dados. A principal característica deste roteador é permitir que usuários em tempo de execução possam definir dinamicamente a utilização de novos campos e protocolos, sem recompilar ou reiniciar o roteador. O roteador proposto foi implementado sobre a plataforma da NetFPGA 1G, utilizando o poder de processamento de um hardware reconfigurável para processar pacotes dinamicamente de redes 1G.

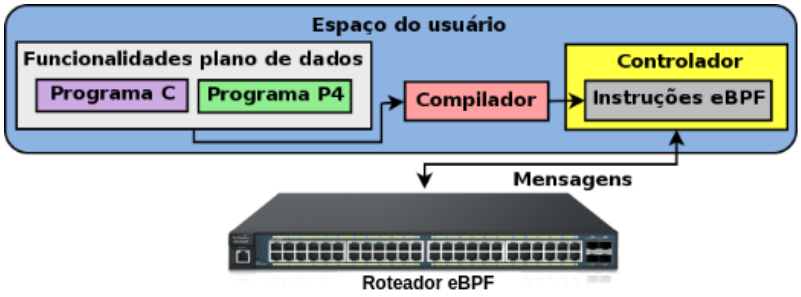


Figura 3.7. Visão geral do roteador eBPF em hardware [Pacífico et al. 2018].

A arquitetura do roteador eBPF (figura 3.7) é composta por um plano de controle (espaço do usuário) e um plano de dados. O plano de controle é formado por um controlador centralizado que envia e instala programas eBPF de aplicações de rede desenvolvidas pelo usuário. Programas eBPF são enviados via conexão *socket* estabelecida entre espaço de usuário e plano de dados. Programas eBPF são compilados e verificados em tempo de execução, e antes de serem instalados no roteador são checados pelo verificador eBPF no espaço do usuário.

O plano de dados do roteador eBPF contém um processador eBPF que realiza operações de análise, casamento e ações dinâmicas de pacotes independente de protocolo utilizando instruções eBPF. O processamento de pacotes no plano de dados do roteador começa quando as instruções eBPF são carregadas na memória de instruções do processador.

A figura 3.8 mostra o caminho de dados do roteador. Os módulos na cor laranja representam os módulos do roteador. Os módulos na cor cinza são os módulos padrão da NetFPGA 1G. À medida que um pacote chega na fila FIFO do módulo *Output Port Lookup*, a (*Finite State Machine* - FSM) retira o pacote da fila e armazena o pacote na memória de dados do processador. Em seguida, o contador de programa (PC) do processador será inicializado, executando as instruções eBPF instaladas. Quando a última instrução for executada o valor do registrador r0 será enviado para o módulo "Ação no pacote". Este módulo é responsável por definir a ação que será realizada no pacote (encaminhamento, descarte ou inundação) quando o pacote for enviado para o módulo *Output Queues*. Após a ação definida, o pacote será retirado da memória de dados, e enviado para *Output Queues*. Em seguida, a FSM espera o próximo pacote para recomeçar o processamento.

Quando PC é inicializado as instruções são buscadas na memória de instruções

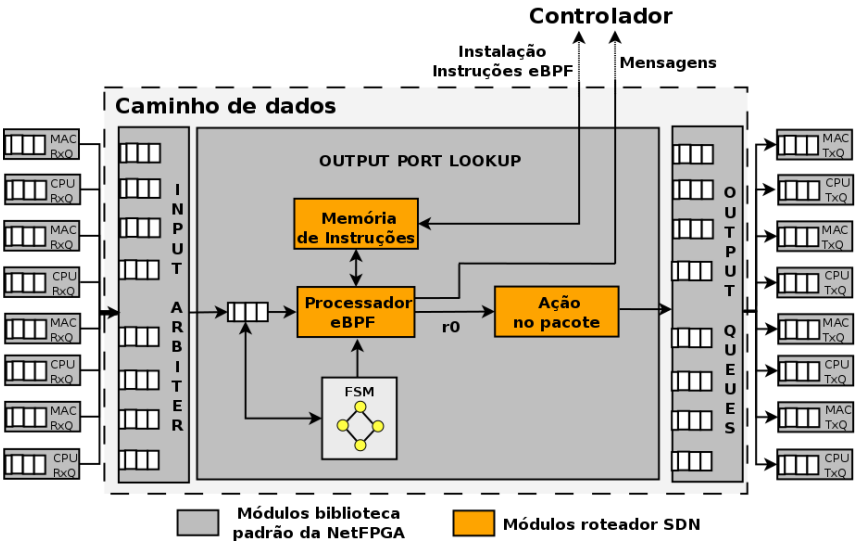


Figura 3.8. Caminho de dados do roteador eBPF implementado na NetFPGA 1G [Pacífico et al. 2018].

de acordo com o endereço atual do PC. Em seguida, as instruções são decodificadas e o valor dos registradores origem e destino são lidos. O próximo passo é executar a operação da instrução na ULA que pode retornar um valor ou um endereço para acesso a memória de dados. Todas as instruções terminam de ser executadas quando ocorre uma escrita na memória de dados ou uma escrita banco de registradores, exceto as instruções de salto (*jump*) que terminam após o retorno da saída da ULA. A figura 3.9 mostra o caminho de dados e controle do processador eBPF.

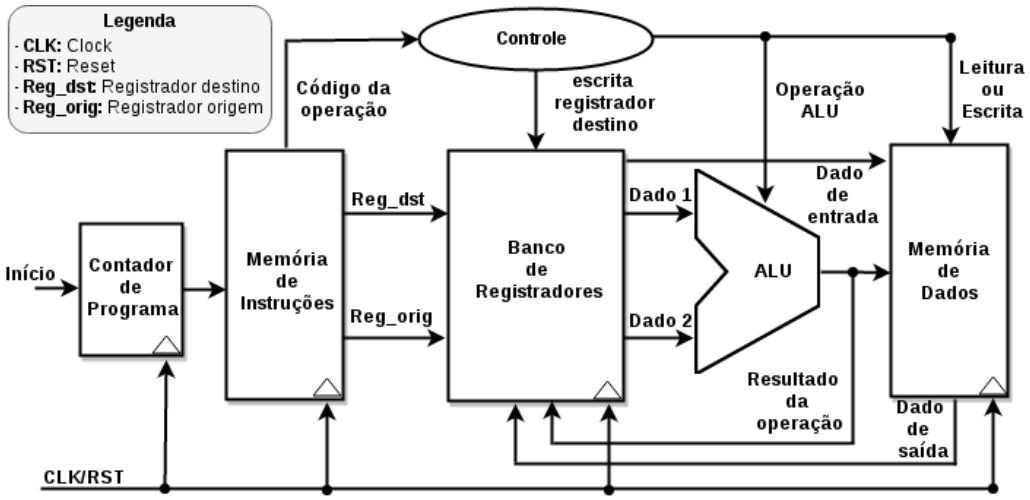


Figura 3.9. Caminho de dados e controle do processador eBPF [Pacífico et al. 2018].

Todos os experimentos foram realizados em um ambiente real composto por dois terminais (A e B) e um controlador conectados no roteador. Os autores validaram o roteador medindo vazão por tamanho de pacote, vazão após reinstalar um programa e vazão trocando entre dois programas distintos (encaminhamento e lista de controle de acesso). Os programas eBPF foram escritos na linguagem C e compilados usando LLVM 3.9 para

plataforma eBPF. A ferramenta *objcopy* foi utilizada para extrair o seguimento no arquivo `elf` referente as instruções. O controlador foi responsável por instalar as instruções no roteador. Em todos os experimentos a ferramenta *iperf* foi usada para criar conexões TCP e UDP entre os terminais. Os resultados obtidos demonstraram que o roteador eBPF tem um plano de dados que permite análise, casamento e ações dinâmicas.

As principais limitações encontrados na implementação do roteador eBPF foram a quantidade de recurso disponível (células lógicas) da NetFPGA 1G e sintetizar instruções como multiplicação, divisão e resto da divisão. Essas instruções são suportadas pelo eBPF, no entanto, a ferramenta da *Xilinx* da NetFPGA 1G não é capaz de sintetizar tais instruções. Os autores descreveram que uma FPGA com quantidade maior de recursos e uma versão de sintetizador recente, por exemplo, NetFPGA SUME usando *Xilinx Vivado*, contornam essas limitações de implementação.

3.8. Funções de rede

Nesta seção são apresentadas algumas funções de redes que demonstram a utilidade e a flexibilidade do eBPF em diferentes plataformas.

3.8.1. Balanceador de carga L4 - l4lb

A Netronome disponibiliza o código de um programa XDP chamado l4lb que implementa um balanceador de carga L4 [Netronome 2019]. O programa l4lb processa os pacotes da rede e calcula um valor hash com base no endereço IP origem, juntamente com as portas TCP ou UDP. O hash gerado é usado como chave em um mapa eBPF.

O mapa eBPF é preenchido com o endereço dos servidores disponíveis para os quais o programa l4lb pode redirecionar o pacote. Este programa estende e insere um cabeçalho IP externo com os dados do mapa. Após o processamento, o programa envia o pacote para o servidor de destino. A computação deste programa pode ser toda realizada pela placa de rede, poupando ciclos para a CPU.

Os scripts para preencher mapas e ler estatísticas são escritos em Python. Esses scripts utilizam o *bpftool* para mostrar como um programa pode ser implementado usando os utilitários *iproute2* e *bpftool*.

3.8.2. RSS Programável

É um escalonador programável de pacotes recebidos para múltiplas CPUs. A técnica de *Receive Side Scaling* (RSS) [Netronome 2019] é utilizada por muitas placas de rede para distribuir a computação dos pacotes para um conjunto de CPUs distintas por meio do uso de múltiplas filas. Porém as implementações geralmente são proprietárias e feitas em hardware, permitindo pouca ou nenhuma programabilidade. Utilizando um programa eBPF, a distribuição dos pacotes pode ser modificada sob demanda por meio de valores associados a mapas ou da substituição completa do programa eBPF carregado.

3.8.3. Monitoramento

Para demonstrar a flexibilidade do BPFabric, os autores implementaram diversas funções de rede de exemplo, disponíveis no repositório oficial do projeto [Jouet and Pezaros 2017b].

Dentre os exemplos estão funções de monitoramento de rede em tempo real. Por se tratar de um dispositivo programável, diferentes tipos de medições podem ser implementadas e instaladas sob demanda no comutador juntamente com a funcionalidade padrão de encaminhamento. Os tipos de medição incluem análise do tempo de chegada entre pacotes, latência média, distribuição do tamanho dos pacotes recebidos e detecção de anomalias.

Por exemplo, a aplicação de análise de tempo de chegada entre pacotes utiliza um mapa do tipo *array* de tamanho 1 para armazenar informações de estado do fluxo de pacotes (tempo de chegada do último pacote, contador de pacotes observados e quantidade de intervalos maiores que o máximo definido) e outro, também do tipo *array*, para armazenar a quantidade de pacotes que chegaram após certos intervalos de tempo de seus predecessores. Quando um pacote chega ao switch, busca-se as informações de estado do fluxo armazenadas no mapa e então calcula-se o intervalo de tempo decorrido desde a chegada do último pacote observado anteriormente. Caso esse tempo seja maior do que o maior intervalo disponível no *array*, então o contador de quantidade de intervalos maiores que o máximo definido é incrementado. Caso contrário, verifica-se qual dos tamanhos de intervalo disponíveis no segundo *array* é o mais próximo do medido e se incrementa e atualiza o valor do contador correspondente. Depois são atualizadas as informações de último pacote recebido e o contador de pacotes observados.

3.8.4. NAT

É possível implementar um tradutor de endereços de rede (NAT) utilizando um programa eBPF e o roteador citado na seção 3.7.2.2. Essa função sobrescreve o endereço IP e porta L4 de origem dos pacotes enviados pelos hospedeiros da rede com um novo par de valores. O mapeamento entre o par original e o novo deve ser armazenado em uma tabela local para posterior consulta. Quando um pacote vindo da Internet é recebido, essa tabela é consultada, utilizando o par *(IP destino, porta destino)* do pacote como chave, e os valores originais são reescritos no pacote. Dessa forma, a rede é capaz de reenviar o pacote para o hospedeiro correspondente. A tabela pode ser implementada utilizando um mapa do tipo `BPF_MAP_TYPE_HASH`, e as operações sobre ela são feitas com as funções auxiliares `bpf_map_lookup` e `bpf_map_update`.

3.8.5. Filtragem de aplicações

Uma operação muito comum feita por administradores de redes é restringir acesso a aplicações de rede específicas. Isso pode ser feito através da filtragem de pacotes com determinados valores de porta de destino nos cabeçalhos de camada de transporte. Por meio do acesso direto ao pacote, um programa eBPF pode processar os cabeçalhos e extrair a porta de destino utilizada, podendo tomar a decisão de efetuar o descarte ou não. Um exemplo dessa aplicação está disponível no repositório deste minicurso [Vieira et al. 2019].

3.8.6. ChaCha

Também é possível implementar uma versão do algoritmo de criptografia ChaCha20 [Nir and Langley 2018] no gancho XDP. Esse algoritmo é utilizado por empresas como o Google [Google 2014] para criptografia simétrica no protocolo *Transport Layer Security* (TLS). Um exemplo de implementação está disponível no repositório [Vieira et al. 2019]. Por conta da limitação do tamanho do código eBPF permitido pelo Linux, o algoritmo

implementado utiliza apenas 8 rodadas, ao invés dos 20 tradicionais.

3.9. Projetos de pesquisa

Nesta seção são descritos projetos de pesquisa que utilizam eBPF para o processamento de pacotes sobre plataformas de software.

3.9.1. Encadeamento de funções de serviço

O encadeamento de funções é um problema recorrente em ambientes de virtualização de funções de redes (*Network Function Virtualization* - NFV). Para oferecer serviços complexos, muitas vezes é necessário combinar diferentes funções (NAT, *proxy*, *firewall*, DPI, etc) em uma ordem pré-determinada, também denominada cadeia. Pacotes de diferentes fluxos podem ser processados por cadeias diferentes com funções e tamanhos distintos, exigindo flexibilidade além da oferecida por mecanismos de roteamento e encaminhamento tradicionais. A *Internet Engineering Task Force* (IETF) propôs uma arquitetura de referência com esse fim [Halpern and Pignataro 2015]. Esse padrão define diferentes elementos de rede que atuam em conjunto sobre um protocolo de encapsulamento especial para habilitar a movimentação dos pacotes pelas funções de uma cadeia na ordem correta. A RFC 8300 [Quinn et al. 2018] define o protocolo *Network Service Header* (NSH), que pode ser usado para esse fim. De forma geral, a implementação desses elementos é feita de forma indireta, através da inclusão de suas ações em dispositivos já existentes, como roteadores e comutadores de rede. Porém, isso cria um alto acoplamento entre a arquitetura de encadeamento e a infraestrutura de rede.

Castanho et al. [Castanho et al. 2019] propõem uma nova arquitetura que visa desacoplar o encadeamento dos dispositivos de rede, denominada *Cadeia-Aberta*. Nela os elementos da RFC 7665 são integrados às funções de rede como estágios de processamento. Cada função de rede é executada por uma máquina virtual ou contêiner individual, na qual os estágios são carregados na forma de programas eBPF. Toda a funcionalidade necessária para habilitar o encadeamento é implementada pelos estágios. Com isso, não há mais a necessidade de elementos intermediários entre as funções para executar o encadeamento, tornando a arquitetura transparente tanto à rede quanto às funções, não exigindo nenhuma modificação nas mesmas e facilitando a sua adoção em ambientes legados.

O processamento é dividido em três estágios: *Dec*, *Enc* e *Fwd*, como mostrado na figura 3.10. Assim que um quadro é recebido, o estágio *Dec* verifica se ele está encapsulado com um cabeçalho NSH. Nesse caso, o encapsulamento de transporte e o NSH são removidos e o pacote é enviado para o espaço de usuário para ser processado pela função de rede (SF). Esse estágio foi implementado no gancho XDP para ter acesso ao pacote no nível mais baixo possível da pilha do *kernel*. Para efetuar o desencapsulamento foi utilizada a função auxiliar `bpf_xdp_adjust_head`:

```
// Remove outer encapsulation + NSH
bpf_xdp_adjust_head(ctx, (int)(outer_header_size + sizeof(struct
  ↪  nshhdr)));
```

As informações do NSH são salvas em um mapa BPF para serem readicionadas ao

pacote durante a transmissão, já no estágio *Enc*. De forma similar ao estágio *Dec*, os estágios *Enc* e *Fwd* precisam modificar o pacote no nível mais baixo possível da rede. Como o gancho XDP está disponível apenas na recepção, o gancho TC foi utilizado para esses dois últimos estágios. No estágio *Enc*, o cabeçalho NSH original é readicionado ao pacote, que é então encaminhado para o estágio *Fwd*. A readição do NSH é possível pois os estágios *Dec* e *Enc* compartilham um mapa de armazenamento temporário de cabeçalhos NSH. Porém, diferentemente do XDP, o gancho TC não apresenta uma função de encapsulamento genérica. Por conta disso os autores utilizaram múltiplas chamadas à função `bpf_skb_vlan_push` para inserir rótulos VLAN que pudessem ser sobrescritos com o encapsulamento desejado:

```
#pragma clang loop unroll(full)
for(int i = 0 ; i < 8 ; i++){
    ret = bpf_skb_vlan_push(...);
    if (ret < 0) {
        return TC_ACT_SHOT;
    }
}
```

Por fim, o estágio *Fwd* utiliza as informações do cabeçalho NSH recém-adicionado para definir a próxima função a ser executada na cadeia. Após uma consulta em um mapa local, o encapsulamento de transporte é alterado com o endereçamento correspondente para que a rede possa encaminhar o pacote para o próximo salto.

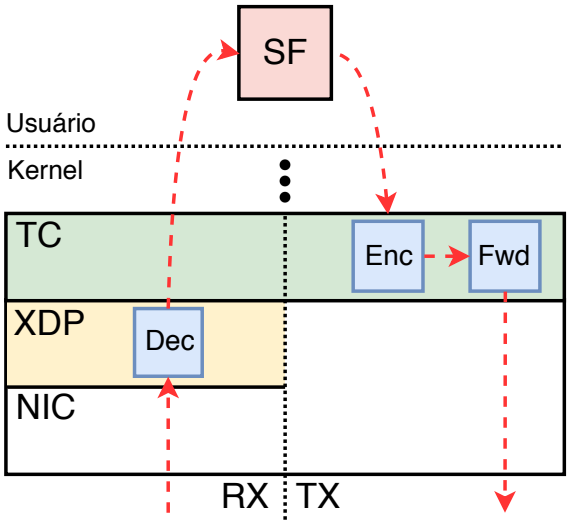


Figura 3.10. Estágios de processamento da arquitetura Cadeia-Aberta implementados como códigos eBPF no *kernel* do Linux [Castanho et al. 2019].

3.9.2. Roteamento por segmento

A demanda recente por maior programabilidade de redes de computadores têm levado ao surgimento de tecnologias como o roteamento por segmento [Filsfils et al. 2018]. Essa técnica permite que administradores de redes especifiquem diferentes ações a serem executadas sobre pacotes em pontos específicos da rede. Utilizando rótulos MPLS ou o protocolo IPv6 com um campo especial chamado *Source Routing Header* (SRH), cada pacote é

encapsulado com uma lista ordenada de ações de roteamento e processamento, denominadas segmentos. Conforme o pacote é movimentado pela rede, dispositivos habilitados processam essa a lista de segmentos e executam as ações especificadas por ela. Isso permite a implementação, por exemplo, de diferentes funções de rede [Abdelsalam et al. 2017].

O *kernel* do Linux já apresenta suporte a roteamento por segmento sobre IPv6 desde a versão 4.10, porém com apenas poucas opções de processamento, como encaminhamento e envio e recebimento de pacotes rotulados. [Duchene et al. 2018] utilizaram o arcabouço eBPF para tornar a criação e especificação de novos segmentos mais genérica e flexível. Por meio da adição de novas funções auxiliares, os autores extenderam o *kernel* do Linux para permitir a implementação de segmentos na forma de programas eBPF. Com isso, novas funções de rede podem ser facilmente desenvolvidas e associadas a regras de roteamento do Linux, facilitando sua integração a ambientes de roteamento por segmento sobre IPv6.

3.9.3. Sketches

Sketches são estruturas de dados probabilísticas compactas que são utilizadas por algoritmos de *streaming* para manter informações resumidas sobre fluxos [Yu et al. 2013]. Suas principais vantagens são: (i) utilização de memória para armazenamento de informações significativamente menor que o volume de dados de entrada, e (ii) *tradeoffs* provados entre a quantidade de memória utilizada e a acurácia obtida. Santos *et al.* [Santos et al. 2019b] propõem e avaliam implementações de aplicações de monitoramento de redes no BPFabric que utilizam *sketches* implementados utilizando eBPF.

A seguir, são apresentados alguns dos *sketches* que podem ser utilizados em diferentes tarefas de monitoramento de rede. Após, descreve-se o que é necessário para implementar *sketches* utilizando o eBPF através do BPFabric.

3.9.3.1. Bloom Filter

O *Bloom Filter* [Bloom 1970] é uma estrutura probabilística que é utilizada para verificar se um elemento pertence ou não a um determinado conjunto. Eficiente em termos de espaço, ela é formada por um *array B* de m bits. Ela também utiliza k funções hash $h_1, h_2, ..., h_k$ para mapear um elemento e para k posições do *array B*. A figura 3.11 apresenta essa estrutura probabilística.

Inicialmente iguais a 0, os bits de B são ativados (se tornam iguais a 1) à medida que novos elementos são inseridos ao conjunto. A inserção de um novo elemento e é feita através da ativação dos bits nas posições $h_1(e), h_2(e), ..., h_k(e)$ de B, ou seja, fazendo-se $B[h_i(e)] = 1$ para $i = 1, 2, ..., k$. Já para determinar se um elemento e_x pertence ao conjunto, é necessário verificar se todos os bits $B[h_i(e_x)]$, para $i = 1, 2, ..., k$, estão ativados. Caso um bit não esteja ativado, este elemento não está no conjunto.

É interessante notar que o Bloom Filter não apresenta falsos negativos, ou seja, ele não acusa a existência de um elemento no conjunto que realmente não faça parte dele. Entretanto, falsos positivos são possíveis. Também destaca-se que a implementação de um Bloom Filter com uma dada uma tolerância a erros, expressa como uma probabilidade p,

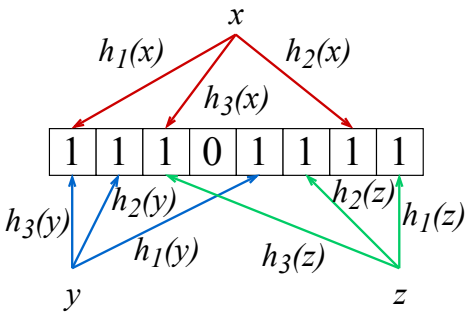


Figura 3.11. Bloom Filter: Elementos *x*, *y* e *z* são mapeados para, por exemplo, 3 posições cada, que indicam quais bits devem ser ativados/verificados.

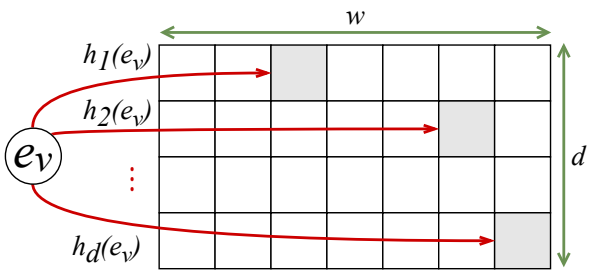


Figura 3.12. Count-Min Sketch: Entrada *e* é mapeada para *d* contadores utilizando *d* funções hash.

e com um número esperado de elementos no conjunto igual a n, requer a utilização de $k = -\log_2 p$ funções hash e de um *array* de tamanho $m = -(n \ln p)/(\ln 2)^2$ bits.

3.9.3.2. Count-Min Sketch

O *Count-Min Sketch* [Cormode and Muthukrishnan 2005] é uma estrutura de dados probabilística compacta que é utilizada para contar a frequência de eventos. Ela é composta por uma matriz M, de tamanho $w \times d$, de contadores que são inicializados com o valor zero. Ela também utiliza um conjunto de funções hash $h_1, h_2, ..., h_d$ independentes par-a-par para mapear um elemento de entrada a um contador em cada uma das linhas da matriz. A figura 3.12 mostra uma representação dessa estrutura de dados.

Existem dois procedimentos fundamentais no Count-Min Sketch: a atualização e a estimativa. O procedimento de atualização registra o acontecimento de um evento e_v através do incremento dos contadores nas posições $M[1,h_1(e_v)], M[2,h_2(e_v)], ..., M[d,h_d(e_v)]$. Já o procedimento de estimativa é utilizado para estimar a frequência de ocorrência do evento e_v como $\hat{f}_e = \min_{i=1,...,d} M[i,h_i(e_v)]$. Observa-se que $\hat{f}_e \geq f_e$, onde f_e é a frequência real do evento e_v. Se $w = \lceil e/\varepsilon \rceil$ e $d = \lceil \ln(1/\delta) \rceil$, então tem-se com probabilidade $1-\delta$ que $\hat{f}_e \leq f_e + \varepsilon N$, onde e é o número de Euler e N é o número total de eventos registrados no sketch.

3.9.3.3. K-ary Sketch

O *K-ary Sketch* é uma estrutura de dados probabilística projetada para estimar a atividade total de uma dada chave em um fluxo [Krishnamurthy et al. 2003]. Similar ao Count-Min Sketch, ela consiste de uma matriz H com dimensões $w \times d$, onde cada linha i é associada a uma função hash h_i. As funções hash utilizadas devem ser 4-universais para que a precisão das estimativas tenham garantias probabilísticas. A figura 3.13 apresenta uma representação dessa estrutura.

Essa estrutura oferece operações para atualizar o *sketch*, estimar um valor atra-

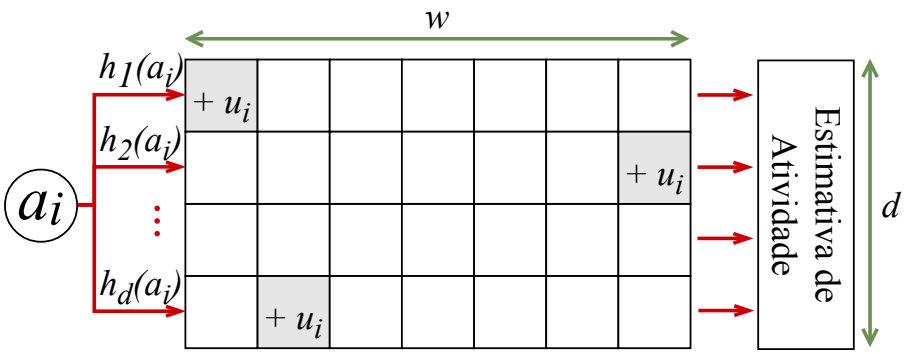


Figura 3.13. K-ary Sketch: Computa-se uma função hash para cada uma das *d* linhas e os contadores associados são incrementados.

vés de uma chave e computar a combinação linear de múltiplos *sketches*. Considerando um fluxo de dados $S = \alpha_1, \alpha_2, ...$, onde $\alpha_i = (a_i, u_i)$ é um par de chave (a_i) e valor de atualização (u_i), a operação de atualização de um item no sketch é dada por $\forall j \in \{1,...,d\}, H[j][h_j(a_i)] = H[j][h_j(a_i)] + u_i$. Já a operação de estimativa para uma chave a_i computa o seguinte valor:

$$mediana_{j \in \{1,...,d\}} \left\{ \frac{H[j][h_j(a_i)] - soma(S)/w}{1 - 1/w} \right\}, \tag{1}$$

onde $soma(S) = \sum_{k \in \{1,...,w\}} H[0][k]$ só precisa ser computado uma vez antes de qualquer estimativa de valores. A operação de combinação une múltiplos sketches fazendo $S = \sum_{l=1}^{n} c_l S_l$, onde c_l é um peso associado ao sketch S_l. Para fazer isto, ela combina cada entrada da matriz H da seguinte forma $H[j][k] = \sum_{l=1}^{n} c_l \times H_l[j][k]$. Considerando esta forma de combiná-los, a estimativa provida pelo k-ary Sketch é estatisticamente sem viés.

3.9.3.4. PCSA

A *Contagem Probabilística com Média Estocástica* (PCSA, do inglês *Probabilistic Counting with Stochastic Averaging*) [Flajolet and Martin 1985] foi proposta com o intuito de estimar o número de elementos distintos em uma grande coleção. Para se ter uma ideia, o PCSA é capaz de contar bilhões de elementos utilizando apenas um número comparavelmente pequeno de bits.

A inserção de um novo elemento ao PCSA utiliza uma função hash e outras duas funções $r(x)$ and $R(x)$. Aqui considera-se que a função hash retorna um inteiro (de 32 ou 64 bits). A função $r(x)$ conta o número de 1's à direita no número de entrada x. Já a função $R(x)$ é definida como $R(x) = 2^{r(x)}$ e é computada utilizando as instruções $\neg x \ \& \ (x+1)$.

O PCSA armazena uma matriz de tamanho $w \times d$, onde w pode ser igual a 32 ou 64 bits e d, o número de linhas na matriz, é utilizado para controlar a precisão dessa estrutura. Quando um novo elemento é adicionado, a função hash é computada e retorna um número k que indica qual a linha da matriz deverá receber o novo dado. Dada a linha k selecionada, executa-se a operação $matriz[k] = matriz[k] \mid R(x)$.

A contagem do número de elementos no PCSA requer a computação da soma $s = \sum_{i=1}^{d} r(matriz[i])$. Esse valor é utilizado para obter o resultado da contagem, que é dado por $d * 2^{(s/d)/.77351}$. A precisão do PCSA, dada uma matriz de $d \times 64$ bits, é igual a $0.78\sqrt{d}$. Esta operação é apresentada no Algoritmo 1.

Algoritmo 1 Operação de contagem do PCSA

```
1: procedure CONTAELEMENTOS(Matriz)
2:     soma = 0.0
3:     for i = 0; i < tamanho(Matriz) do
4:         soma+ = r(Matriz[i])
5:     end for
6:     media = soma/tamanho(Matriz)
       return tamanho(Matriz) * 2^(media/.77351)
7: end procedure
```

3.9.3.5. Implementação de sketches utilizando eBPF

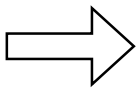
É possível implementar sketches para o eBPF no BPFabric a partir da definição de novos tipos de mapas (seção 3.5.2). De forma geral, para criar um novo mapa é necessário (i) definir um novo tipo que será utilizado para definir os parâmetros do novo mapa, (ii) a nova estrutura de dados que comporta essa estrutura, e (iii) definir as operações desse novo tipo. Exemplos de implementações dos sketches mencionados anteriormente podem ser obtidos em [Santos et al. 2019a].

A definição de um novo tipo de mapa é feita através da inserção de uma nova entrada no *enum* `bpf_map_type`, localizada no arquivo *bpfmap.h*. Por exemplo, a definição de um novo tipo correspondente ao Bloom Filter pode ser feita através da adição do identificador `BPF_MAP_TYPE_BITMAP` ao final do `bpf_map_type`:

```
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC,
    BPF_MAP_TYPE_HASH,
    BPF_MAP_TYPE_ARRAY,
    BPF_MAP_TYPE_BITMAP,
};
```

Além disso, como apresentado na seção 3.5.2, os parâmetros de criação de uma instância de um mapa são definidos pela *struct* de configuração `bpf_map_def`. Essa *struct* contém atributos que definem o tipo do mapa (*map_type*), tamanho da chave, tamanho dos valores armazenados, dentre outros necessários para a criação. Como cada sketch possui um conjunto de parâmetros diferentes, é possível adicionar ou modificar campos da `bpf_map_def` para que ela comporte parâmetros específicos. No BPFabric, isso pode ser feito editando o arquivo *includes/ebpf_switch.h*. Seguindo com o exemplo do Bloom Filter, seria possível modificar a *struct* para adicionar parâmetros como a quantidade de funções hash (k) e a quantidade de bits (m) do *array*. Uma forma de fazer isso é produzir a seguinte modificação:

```
struct bpf_map_def {
    unsigned int type;
    unsigned int key_size;
    unsigned int value_size;
    unsigned int max_entries;
    unsigned int map_flags;
};
```



```
struct bpf_map_def {
    unsigned int type;
    union{
        unsigned int key_size;
        unsigned int num_hashes;
    };
    union{
        unsigned int value_size;
        unsigned int num_bits;
    };
    unsigned int max_entries;
    unsigned int map_flags;
};
```

Para definir as operações de um novo tipo é necessário implementar um conjunto de operações básicas definidas no BPFabric. Algumas delas são: `map_alloc`, responsável pela criação do mapa e alocação de memória; `map_free` responsável por liberar a memória utilizada no mapa; `map_lookup_elem`, utilizado para pesquisar um o valor associado a uma chave em um mapa; `map_get_next_key`, utilizado para retornar a próxima chave de um mapa; `map_update_elem`, utilizado para atualizar o valor associado a uma chave; `map_delete_elem`, utilizado para deletar o valor associado a uma chave. A implementação dessas operações deve ocorrer em um par de arquivos .c e .h (por exemplo, *bitmap.c* e *bitmap.h*) e as assinaturas das funções implementadas devem ser correspondidas às referências do BPFabric utilizando o vetor de operações de mapas `bpf_map_types`, disponível no arquivo *bpfmap.c*. Vale notar que também é necessário incluir nesse arquivo uma referência para o arquivo `header` com as definições das operações do novo tipo.

Considerando o exemplo de implementação de um Bloom Filter, pode-se definir as seguintes funções: `bitmap_map_alloc`, que fica responsável por criar e inicializar a matriz que será utilizada pelo bitmap; `bitmap_map_free` que libera a memória alocada pelo bitmap; `bitmap_map_lookup_elem` que recebe uma chave e retorna uma variável indicando se a chave se encontra no sketch utilizando as operações apresentadas na seção 3.9.3.1; `bitmap_map_update_elem` que recebe uma chave e uma flag, sendo a que flag possibilita escolher se a função deve limpar o bitmap ou adicionar um novo elemento a ele. Funções como `bitmap_map_delete_elem` e `bitmap_map_get_next_key` podem ser definidas mas não precisam ser implementadas, já que não existem operações no sketch que possam ser associadas a essas funções. Um exemplo de como ficaria o vetor `bpf_map_types` após a adição das referências das operações implementadas Bloom Filter (identificadas como `bitmap_*`) pode ser visto abaixo.

```
const struct bpf_map_ops bpf_map_types[] = {
    [BPF_MAP_TYPE_HASH] = {
        .map_alloc = htab_map_alloc,
        .map_free = htab_map_free,
```

```
        .map_get_next_key = htab_map_get_next_key,
        .map_lookup_elem = htab_map_lookup_elem,
        .map_update_elem = htab_map_update_elem,
        .map_delete_elem = htab_map_delete_elem,
    },
    [BPF_MAP_TYPE_ARRAY] = {
        .map_alloc = array_map_alloc,
        .map_free = array_map_free,
        .map_get_next_key = array_map_get_next_key,
        .map_lookup_elem = array_map_lookup_elem,
        .map_update_elem = array_map_update_elem,
        .map_delete_elem = array_map_delete_elem,
    },
    [BPF_MAP_TYPE_BITMAP] = {
        .map_alloc = bitmap_map_alloc,
        .map_free = bitmap_map_free,
        .map_get_next_key = bitmap_map_get_next_key,
        .map_lookup_elem = bitmap_map_lookup_elem,
        .map_update_elem = bitmap_map_update_elem,
        .map_delete_elem = bitmap_map_delete_elem,
    }
};
```

Existem duas considerações que podem ser feitas ainda na parte de definição de operações. A primeira delas é que é possível definir novas operações para os mapas. Se existe alguma operação no sketch que não corresponde a uma operações já definidas, uma nova operação pode ser adicionada. Essa adição pode ser feita através da inclusão de um novo campo na estrutura de operações `bpf_map_ops`, disponível no arquivo bpfmap.h. A segunda consideração é que nem todas as operações definidas necessitam ser implementadas para todos os mapas. Por exemplo, a operação *map_get_next_key* não tem uma função para alguns sketches, podendo assim não ser implementada.

3.9.4. Outros projetos

Além dos já citados, há vários outros projetos que fazem uso do arcabouço eBPF para adicionar programabilidade ao plano de dados em diferentes contextos. [Ahmed et al. 2016] utilizam programas eBPF para modificar o caminho de dados de redes virtuais de centros de dados. [Jouet et al. 2015] e [Tu et al. 2017] propõem extensões ao protocolo OpenFlow e ao comutador virtual Open vSwitch, respectivamente, utilizando programas eBPF. [Bertrone et al. 2018] propõem uma nova versão da ferramenta *iptables* utilizando a tecnologia.

Em cenários de IoT e no paradigma de computação de borda, as mesmas informações de um sensor podem ser usadas por vários aplicativos em locais diferentes. Neste caso, o fluxo de dados precisa ser replicado. Em [Baidya et al. 2018], o fluxo de tráfego e replicação de pacotes para cada processo de computação específico é controlado por um programa eBPF.

Algumas empresas já utilizam eBPF em ambientes de produção na indústria, como é o caso da Cloudflare, que utiliza XDP em seu sistema de mitigação de ataques de negação de serviço [Bertin 2017], e do Facebook, que desenvolveu um balanceador de carga baseado em eBPF chamado Katran [Facebook 2018].

Por fim, projetos de código aberto baseados em eBPF também tem surgido nos últimos anos. Cilium [Cilium 2018] é um projeto para prover segurança em redes de contêineres e aplicações com microserviços. O projeto IOVisor [IOvisor 2019] mantém múltiplos sub-projetos baseados em eBPF como *gobpf* [gobpf 2019], que permite a interação com o sistema eBPF utilizando a linguagem Go, as ferramentas *ply* [ply 2019] e *bpftrace* [bpftrace 2019] para introspecção do kernel, além do BCC [BCC 2019a] e do *ubpf* [ubpf 2019], já discutidos anteriormente.

3.10. Desafios e limitações

A tecnologia eBPF habilita flexibilidade no processamento de pacotes por permitir a execução de códigos de modo dinâmico sobre o *kernel* sem instalar qualquer módulo extra. Essa tecnologia também suporta cadeia de serviços arbitrários (conjunto de funções de rede conectadas) e integração com XDP para ter acesso antecipado ao pacote. Tais características em conjunto são difíceis de ser encontradas em outros sistemas. eBPF também é conhecido por suas limitações devido às restrições da VM eBPF, responsáveis por garantir a integridade do sistema. Nesta seção são descritos os desafios e as limitações da tecnologia eBPF com base na análise realizada por [Miano et al. 2018].

Número de instruções limitado: 4096 é o número máximo de instruções que um programa eBPF pode ter para garantir que o programa termine de ser executado dentro do *kernel*. Essa restrição pode ser um problema quando funções de redes complexas são implementadas, pois o número de instruções geradas após o código C ser compilado pode ser maior que 4096.

Uma maneira de contornar essa limitação consiste dividir um programa com número de instruções maior que 4096 em vários subprogramas, e saltar de um subprograma para outro usando chamadas de calda (*tail calls*). Essa técnica habilita o desenvolvimento de serviços de rede como uma coleção de módulos fracamente acoplados, onde cada módulo realiza uma função diferente (análise, classificação ou modificação de campos) sem *overhead* ao saltar de um módulo para outro. Oito chamadas são o número máximo de chamadas subsequentes que não afetam o desempenho do eBPF.

Laço infinito: Programas eBPF antes de serem carregados no *kernel* são checados pelo verificador para garantir que programas não possam prejudicar o sistema. O verificador procura múltiplas ameaças (laço infinito, salto para trás, acesso a endereços inválidos, etc.). Quando uma ameaça é detectada, o programa automaticamente é rejeitado. Laços são proibidos em programas eBPF, pois um programa pode não terminar de ser executado dentro do *kernel*.

eBPF contorna essa limitação explorando o uso de diretivas `pragma unroll` do compilador LLVM, reescrevendo um laço como uma sequência repetida de instruções independentes. Essa técnica soluciona o problema. No entanto, ela aumenta o número de instruções, e para alguns casos pode não funcionar por não conseguir definir o valor da constante superior do término do laço, por exemplo, número máximo de cabeçalhos IPv6, rótulo MPLS alinhado e tamanho do pacote. Listamos três exemplos de aplicações no qual o uso do laço pode ser um problema:

1. **Análise de cabeçalhos alinhados:** No protocolo IPv6 percorrer todos os cabeça-

lhos de extensão é necessário para encontrar o último cabeçalho que indica o tipo do protocolo da camada superior na carga do pacote. Esse mesmo problema ocorre nos cabeçalhos MPLS e VLAN, no qual o número de instâncias não é conhecido a priori. Desenvolver funções de rede que realizam essas ações no eBPF não são possíveis de serem implementados sem introduzir restrições adicionais.

2. **Processamento da carga do pacote:** eBPF não permite escanear todo o pacote. No entanto, essa funcionalidade é requerida, por exemplo, para verificar a presença de uma assinatura na carga do pacote. Existem casos que percorrer a carga do pacote pode ser evitado devido ao uso de cabeçalhos específicos disponíveis no pacote que executam todo o trabalho, por exemplo, recalcular o *checksum* L3/L4.

3. **Busca linear estruturas de dados:** Algoritmos utilizam busca linear para encontrar determinado conteúdo em estruturas de dados. No eBPF, mapas são exemplos de estruturas de dados. Essas estruturas precisam ser adaptadas para permitir busca linear de um conteúdo. *Firewall* é um exemplo de aplicação que utiliza busca linear em um mapa para encontrar a regra que corresponde ao pacote. Neste exemplo, todas as políticas ativas são escaneadas através da busca linear.

Enviar o mesmo pacote para múltiplas portas: Requisições ARP, *multicast* e inundação são exemplos de funções de rede que precisam enviar o mesmo pacote para várias portas simultaneamente. No entanto, três problemas podem ser encontrados quando essa operação é implementada no eBPF. Primeiro, todas as interfaces precisam ser percorridas para encaminhar o pacote de acordo com número de vezes desejado. Essa operação pode apenas ser implementada se for possível definir o valor da constante do término do laço. Segundo, o pacote precisa ser clonado antes de ser enviado por uma interface adicional. Essa operação pode ser realizada utilizando a função auxiliar `bpf_skb_clone_redirect()`, duplicando simultaneamente e encaminhando o pacote original para interface adicional. No entanto, não existe uma função similar para programas XDP. Essa função está disponível apenas quando o programa eBPF está anexado ao gancho TC. Terceiro, se o serviço faz parte de uma cadeia virtual (*virtual chain*) composta por várias funções de rede virtuais conectadas por várias chamadas subsequentes, tal abordagem falha. Isso ocorre porque a função de redirecionamento segue por uma chamada subsequente, e nunca retorna o controle para o chamador, bloqueando o código do chamador de enviar pacote para várias portas.

Processamento de pacote dirigido a evento: A execução de um programa eBPF no *kernel* é acionada por evento. O único evento suportado por programas TC/XDP ocorre quando o quadro transita sobre um gancho selecionado dentro do *kernel*. Essa característica previne o plano de dados eBPF de reagir a outros eventos, tal como, tempo expirado de um pacote que sinaliza a necessidade de enviar outro pacote periodicamente, ou atualizar uma entrada que expirou na tabela. Tais eventos devem ser processados em outro lugar, por exemplo, no plano de controle ou no módulo caminho lento (*slow path*).

O módulo caminho lento é um novo componente que pode ser introduzido no plano de controle para executar código arbitrário, e lida com situações nas quais a versão atual do eBPF não pode ser utilizada. A funcionalidade deste módulo será receber pacotes

enviados do plano de dados eBPF, reagir de acordo com processamento arbitrário definido pelo desenvolvedor e encaminhar o pacote modificado para porta de saída.

Não suportar plano de controle complexo: eBPF apresenta um plano de controle simples composto por um conjunto de abstrações que ajudam desenvolvedores a criar código no plano de dados (ganchos, mapas, etc). Como consequência disso, *frameworks* eBPF existentes são primitivos em relação à tarefa de controle. Serviços de rede são diferentes de operações no espaço do usuário e mapas, requerendo um plano de controle mais sofisticado, que suporte o módulo caminho lento ou processe pacotes especiais (por exemplo, protocolos de roteamento). A simplicidade do plano de controle eBPF obriga desenvolvedores a dedicar horas no desenvolvimento de códigos de serviços rede por não terem um *framework* eBPF que auxilia no desenvolvimento de aplicações.

3.11. Conclusão

Neste trabalho apresentamos uma visão dos aspectos teóricos e práticos no desenvolvimento de pesquisa relacionado ao processamento rápido de pacotes. Na parte teórica, apresentamos as máquinas BPF e eBPF, a visão geral do sistema, ganchos e resultados de pesquisa recente. Na parte prática, focamos no eBPF e no gancho XDP. Mostramos como utilizá-los, exemplos e ferramentas. Dado os seus potencias no processamento rápido de pacotes, consideramos que existe um grande potencial no desenvolvimento de novos projetos de pesquisa com eBPF e XDP, seja como ferramenta para o desenvolvimento de novas funções de rede, seja permitindo prover novas funcionalidades no plano de dados, seja como ferramenta para o desenvolvimento de novos padrões e protocolos de comunicação, seja no desenvolvimento de novos protótipos de pesquisa, seja como alvo de estudos sobre novas soluções de rede. Certamente, eBPF e XDP ajudarão no desenvolvimento de novos trabalhos interessantes e com grande potencial na área de redes de computadores. Com certeza, mais trabalhos interessantes na área estão por vir.

Referências

[Abdelsalam et al. 2017] Abdelsalam, A., Clad, F., Filsfils, C., Salsano, S., Siracusano, G., and Veltri, L. (2017). Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure. In *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*, pages 1–5. IEEE.

[Ahmed et al. 2016] Ahmed, Z., Alizai, M. H., and Syed, A. A. (2016). Inkev: Inkernel distributed network virtualization for dcn. *ACM SIGCOMM Computer Communication Review*, 46(3).

[Baidya et al. 2018] Baidya, S., Chen, Y., and Levorato, M. (2018). ebpf-based content and computation-aware communication for real-time edge computing. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WORKSHOPS)*, pages 865–870.

[BCC 2019a] BCC (2019a). BPF Compiler Collection. `https://github.com/iovisor/bcc`.

[BCC 2019b] BCC (2019b). BPF program types. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md\#program-types.

[BCC 2019c] BCC (2019c). Compiling P4 to EBPF. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4.

[BCC 2019d] BCC (2019d). XDP compatible drivers. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp.

[Beckett et al. 2018] Beckett, D., Joubert, J., and Horman, S. (2018). Host dataplane acceleration (hda).

[Bertin 2017] Bertin, G. (2017). Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2.

[Bertrone et al. 2018] Bertrone, M., Miano, S., Risso, F., and Tumolo, M. (2018). Accelerating Linux Security with eBPF Iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, SIGCOMM '18, pages 108–110, New York, NY, USA. ACM.

[Bloom 1970] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

[Bosshart et al. 2014] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

[bpftool 2018] bpftool, A. (2018). Manual bpftool. https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/bpf/bpftool/Documentation/bpftool.rst.

[bpftrace 2019] bpftrace (2019). High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace.

[Budiu 2015] Budiu, M. (2015). Compiling p4 to ebpf.

[Castanho et al. 2019] Castanho, M. S., Dominicini, C. K., and Vieira, M. A. M. (2019). Cadeia-Aberta: Arquitetura para SFC em Kernel usando eBPF. In *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, Porto Alegre, RS, Brasil. SBC.

[Cilium 2018] Cilium (2018). Cilium 1.0: Bringing the BPF Revolution to Kubernetes Networking and Security. https://cilium.io/blog/2018/04/24/cilium-10/.

[Cormode and Muthukrishnan 2005] Cormode, G. and Muthukrishnan, S. (2005). An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75.

[Dargahi et al. 2017] Dargahi, T., Caponi, A., Ambrosin, M., Bianchi, G., and Conti, M. (2017). A survey on the security of stateful sdn data planes. *IEEE Communications Surveys & Tutorials*, 19(3):1701–1725.

[Duchene et al. 2018] Duchene, F., Jadin, M., and Bonaventure, O. (2018). Exploring various use cases for ipv6 segment routing. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, SIGCOMM '18, pages 129–131, New York, NY, USA. ACM.

[Dumazet 2011] Dumazet, E. (2011). A jit for packet filters. `https://lwn.net/Articles/437981/`.

[Facebook 2018] Facebook (2018). Katran source code repository. `https://github.com/facebookincubator/katran`.

[Feamster et al. 2014] Feamster, N., Rexford, J., and Zegura, E. (2014). The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98.

[Filsfils et al. 2018] Filsfils, C., Previdi, S., Ginsberg, L., Decraene, B., Litkowski, S., and Shakir, R. (2018). Segment routing architecture. RFC 8402, RFC Editor.

[Flajolet and Martin 1985] Flajolet, P. and Martin, G. N. (1985). Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209.

[gobpf 2019] gobpf (2019). Go bindings for creating BPF programs. `https://github.com/iovisor/gobpf`.

[Google 2014] Google (2014). Speeding up and strengthening HTTPS connections for Chrome on Android. `https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html`.

[Halpern and Pignataro 2015] Halpern, J. and Pignataro, C. (2015). Service function chaining (SFC) architecture. RFC 7665, IETF.

[Høiland-Jørgensen et al. 2018] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. (2018). 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*, CoNEXT '18, pages 54–66, New York, NY, USA. ACM.

[IOvisor 2019] IOvisor (2019). Iovisor project. www.iovisor.org. Disponível em 29/03/2019.

[Jouet et al. 2015] Jouet, S., Cziva, R., and Pezaros, D. P. (2015). Arbitrary packet matching in openflow. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6.

[Jouet and Pezaros 2017a] Jouet, S. and Pezaros, D. P. (2017a). Bpfabric: Data plane programmability for software defined networks. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '17, pages 38–48, Piscataway, NJ, USA. IEEE Press.

[Jouet and Pezaros 2017b] Jouet, S. and Pezaros, D. P. (2017b). BPFabric implementations. `https://github.com/UofG-netlab/BPFabric`.

[Krishnamurthy et al. 2003] Krishnamurthy, B., Sen, S., Zhang, Y., and Chen, Y. (2003). Sketch-based change detection: Methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC '03, pages 234–247, New York, NY, USA. ACM.

[libbpf 2018] libbpf, A. (2018). libbpf source code. `https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/lib/bpf`.

[Maguire 2019] Maguire, A. (2019). Notes on bpf (1) - a tour of progam types. https://blogs.oracle.com/linux/notes-on-bpf-1.

[McCanne and Jacobson 1993] McCanne, S. and Jacobson, V. (1993). The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA. USENIX Association.

[McKeown et al. 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

[Miano et al. 2018] Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M. V., and Tumolo, M. (2018). Creating Complex Network Services with eBPF: Experience and Lessons Learned. *High Performance Switching and Routing (HPSR). IEEE*, pages 1–8.

[Mijumbi et al. 2016] Mijumbi, R., Serrat, J., Gorricho, J. L., Bouten, N., De Turck, F., and Boutaba, R. (2016). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials*, 18(1):236–262.

[Monnet 2019] Monnet, Q. (2019). Rust virtual machine and JIT compiler for eBPF programs. `https://github.com/qmonnet/rbpf`.

[Netronome 2019] Netronome (2019). Sample bpf offload apps. `https://github.com/Netronome/bpf-samples`.

[Nir and Langley 2018] Nir, Y. and Langley, A. (2018). ChaCha20 and Poly1305 for IETF Protocols. RFC 8439.

[Pacífico et al. 2018] Pacífico, R. D. G., Coelho, G. R., Vieira, M. A. M., and Nacif, J. A. M. (2018). Roteador SDN em hardware independente de protocolo com análise, casamento e ações dinâmicas. In *Anais do XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, Porto Alegre, RS, Brasil. SBC.

[ply 2019] ply (2019). Dynamic Tracing in Linux. https://github.com/iovisor/ply.

[Quinn et al. 2018] Quinn, P., Elzur, U., and Pignataro, C. (2018). Network service header (NSH). RFC 8300, IETF.

[Santos et al. 2019a] Santos, E. R. S., Câmara Júnior, E. P. M., and Vieira (2019a). Bpfabric implementing sketches. https://github.com/elerson/BPFabric.

[Santos et al. 2019b] Santos, E. R. S., Câmara Júnior, E. P. M., Vieira, M. A. M., and Vieira, L. F. M. (2019b). Aplicações de monitoramento de tráfego utilizando redes programáveis eBPF. In *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, Porto Alegre, RS, Brasil. SBC.

[Schulist et al. 2019] Schulist, J., Borkmann, D., and Starovoitov, A. (2019). Linux socket filtering aka berkeley packet filter (bpf). www.kernel.org/doc/Documentation/networking/filter.txt. Disponível em 17/03/2019.

[Sharma et al. 2017] Sharma, N. K., Kaufmann, A., Anderson, T. E., Krishnamurthy, A., Nelson, J., and Peter, S. (2017). Evaluating the power of flexible packet processing for network resource allocation. In *NSDI*, pages 67–82.

[Song 2013] Song, H. (2013). Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132. ACM.

[Tu et al. 2017] Tu, C., Stringer, J., and Pettit, J. (2017). Building an extensible open vswitch datapath. *SIGOPS Oper. Syst. Rev.*, 51(1):72–77.

[ubpf 2019] ubpf (2019). Userspace eBPF VM. https://github.com/iovisor/ubpf.

[Vieira et al. 2019] Vieira, M. A. M., Pacífico, R. D. G., Castanho, M. S., Santos, E. R. S., Câmara Júnior, E. P. M., and Vieira, L. F. M. (2019). Tutorial eBPF. https://github.com/mscastanho/bpf-tutorial.

[VMWare 2018] VMWare (2018). p4c-xdp. https://github.com/vmware/p4c-xdp.

[Wang et al. 2017] Wang, H., Soulé, R., Dang, H. T., Lee, K. S., Shrivastav, V., Foster, N., and Weatherspoon, H. (2017). P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, pages 122–135. ACM.

[Yu et al. 2013] Yu, M., Jose, L., and Miao, R. (2013). Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42.

[Zilberman et al. 2015] Zilberman, N., Watts, P. M., Rotsos, C., and Moore, A. W. (2015). Reconfigurable network systems and software-defined networking. *Proceedings of the IEEE*, 103(7):1102–1124.