

Processamento Rápido de Pacotes com eBPF e XDP

Alunos¹: Racyus Delano, Matheus Castanho,
Eduardo Câmara e Elerson Santos.

Professores²: Marcos A. M. Vieira e Luiz F. M. Vieira

Emails¹: {racyus,matheus.castanho,epmcj,elerson}@dcc.ufmg.br
Emails²: {mmvieira, lfvieira}@dcc.ufmg.br

Apresentação

Belo Horizonte, Campus UFMG e DCC

Belo Horizonte - Savassi



Praça da Liberdade



Praça Diogo de
Vasconcelos



Edifício Oscar Niemeyer

Belo Horizonte - Região Centro-Sul



Feira Hippie



Serra do Curral

Belo Horizonte - Mercado Central



Belo Horizonte - Cidade Administrativa



Prédios projetados por Niemayer
Dois edifícios:
Minas (frente) e Gerais (atrás)



Palácio Tiradentes
(Palácio do Governador)
Prédio é suspenso pelo teto

Belo Horizonte - Pampulha



Museu de Arte Moderna



Parque Guanabara, Mineirinho e Mineirão



Lagoa da Pampulha

Campus UFMG - Área Central



Campus UFMG - Área Central



Reitoria

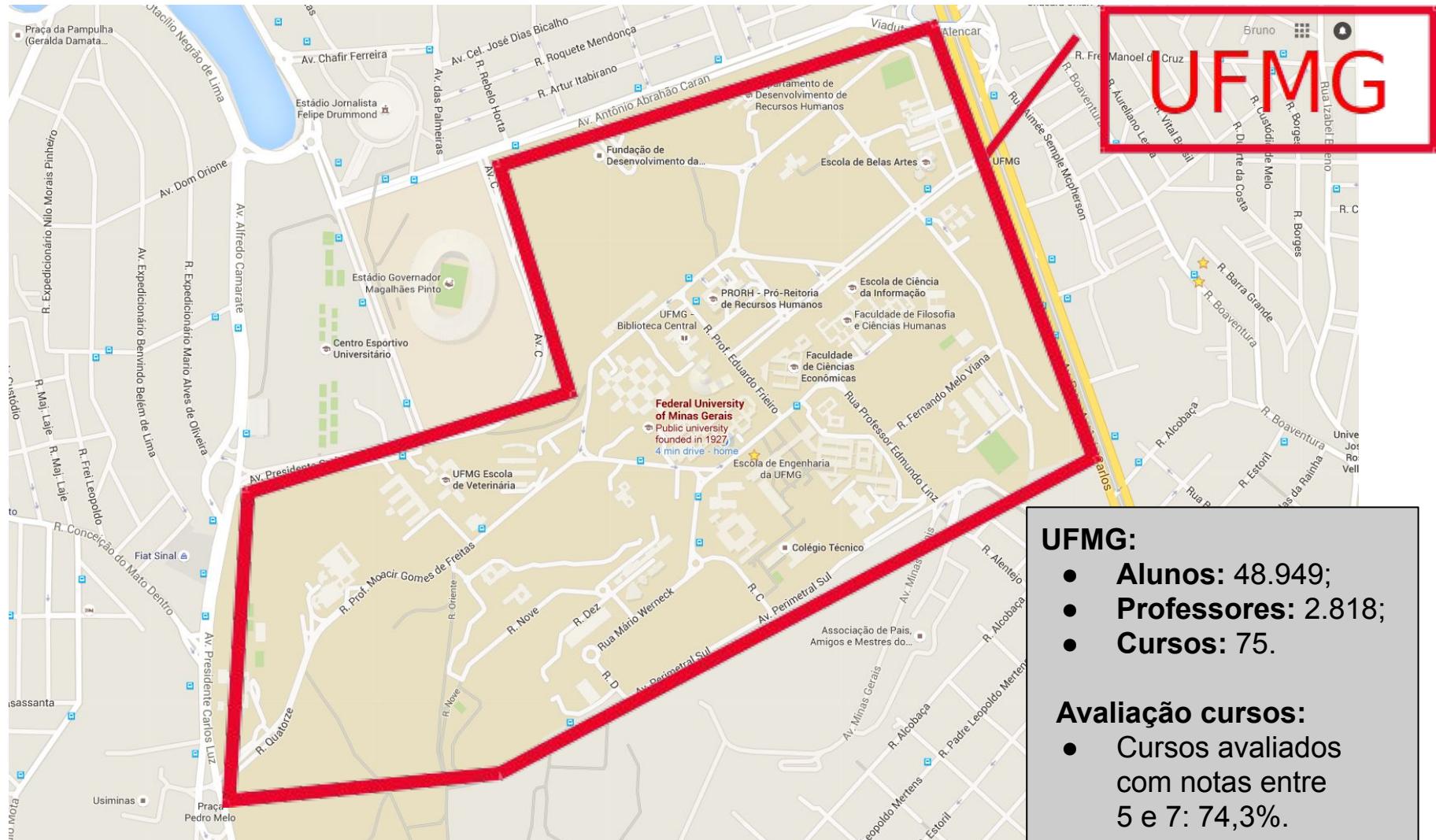


Biblioteca Central



Praça de serviços

Campus UFMG - Dados



UFMG:

- **Alunos:** 48.949;
- **Professores:** 2.818;
- **Cursos:** 75.

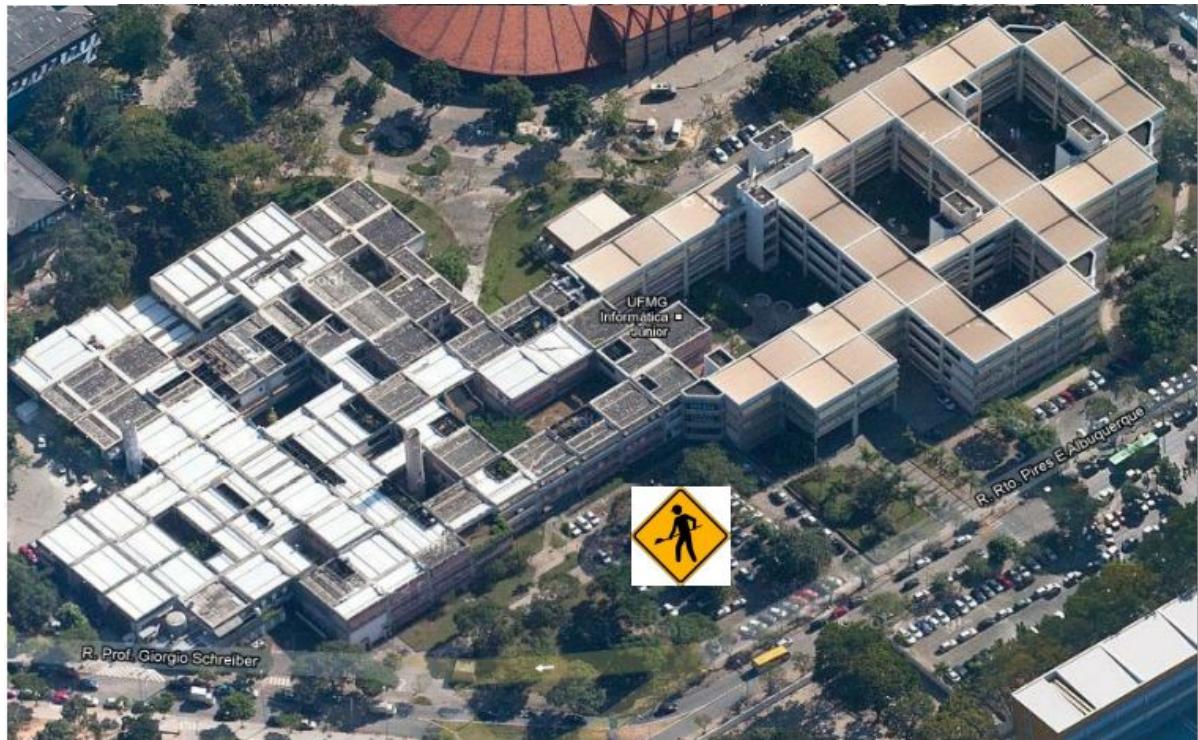
Avaliação cursos:

- Cursos avaliados com notas entre 5 e 7: 74,3%.

Departamento de Ciência da Computação-DCC



**Novo prédio da
Ciência da Computação**



Departamento de Ciência da Computação-DCC

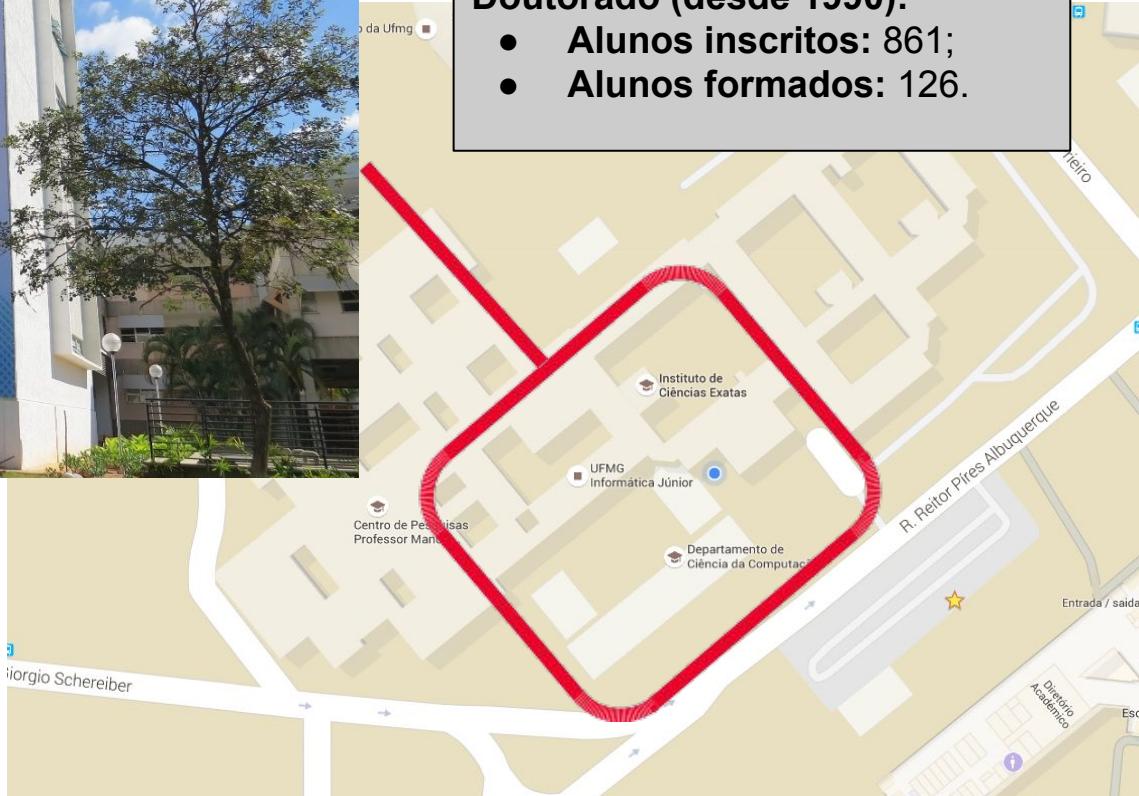


Mestrado (desde 1974):

- **Alunos inscritos:** 137;
- **Alunos formados:** 861.

Doutorado (desde 1990):

- **Alunos inscritos:** 861;
- **Alunos formados:** 126.



Departamento de Ciência da Computação (DCC)

CSRankings: Computer Science Rankings

CSRankings is a metrics-based ranking of top computer science institutions around the world. Click on a triangle (▶) to expand areas or institutions. Click on a name to go to a faculty member's home page. Click on a pie (the 🥧 after a name or institution) to see their publication profile as a pie chart. Click on a Google Scholar icon (ⓘ) to see publications, and click on the DBLP logo (DOI) to go to a DBLP entry.

Rank institutions in by publications from to

All Areas [\[off\]](#) [\[on\]](#)

AI [\[off\]](#) [\[on\]](#)

- ▶ Artificial intelligence
- ▶ Computer vision
- ▶ Machine learning & data mining
- ▶ Natural language processing
- ▶ The Web & information retrieval

Systems [\[off\]](#) [\[on\]](#)

- ▶ Computer architecture
- ▶ Computer networks
- ▶ Computer security
- ▶ Databases
- ▶ Design automation
- ▶ Embedded & real-time systems
- ▶ High-performance computing
- ▶ Mobile computing
- ▶ Measurement & perf. analysis
- ▶ Operating systems
- ▶ Programming languages
- ▶ Software engineering

Theory [\[off\]](#) [\[on\]](#)

- ▶ Algorithms & complexity
- ▶ Cryptography
- ▶ Logic & verification

#	Institution	Count	Faculty
1	▶ UFMG 	1.7	29
2	▶ UFRGS 	1.4	25
3	▶ PUC-RIO 	1.2	13
3	▶ Universidad de Chile 	1.2	9
5	▶ FURG Federal University of Rio Grande 	1.1	4
5	▶ PUC-RS 	1.1	4
5	▶ UFF 	1.1	8
5	▶ UFPE 	1.1	6
5	▶ UNICAMP 	1.1	12
5	▶ USP 	1.1	6
5	▶ USP-ICMC 	1.1	13
5	▶ Universidade Federal de Viçosa 	1.1	2
5	▶ University of Buenos Aires 	1.1	9
14	▶ UFMS 	1.0	3
14	▶ UFRJ 	1.0	4
14	▶ UFU 	1.0	2
14	▶ Universidad de los Andes 	1.0	2
14	▶ Universidade de Brasília 	1.0	2

Fonte: [CSRankings](#)

Roteiro

- Contextualização eBPF;
- Tipos de ganchos e primeiros passos;
- Intervalo;
- Interagindo com eBPF do espaço de usuário;
- Gancho *Traffic Control* (TC);
- Expandindo eBPF com BPFabric;
- Ferramentas;
- Funções de rede e projetos de pesquisa;
- Desafios e limitações eBPF;
- Conclusão.

Considerações iniciais

- Material disponível no site do minicurso:



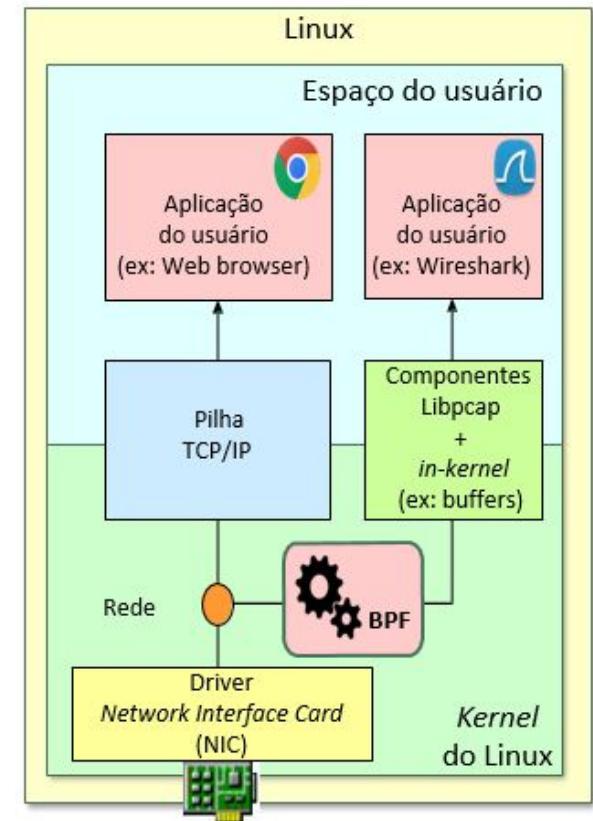
tiny.cc/minicurso-ebpf

- **Aviso:** durante a apresentação, caminhos começando com *linux* se referem ao código fonte do *kernel*;
- **Perguntas são bem-vindas!**

Contextualização eBPF

Berkeley Packet Filter (BPF)

- Introduzido no *kernel* do Linux 2.1.75 (1997);
- Inicialmente usado como filtro de pacotes pela ferramenta de captura tcpdump (via libpcap);
- Execução de programas dentro do *kernel* de modo genérico;
 - Chamadas de sistema sem *overhead*;
 - Troca de contexto entre *kernel* e usuário;
- CPU virtual baseado a evento;
 - Pacotes da rede.

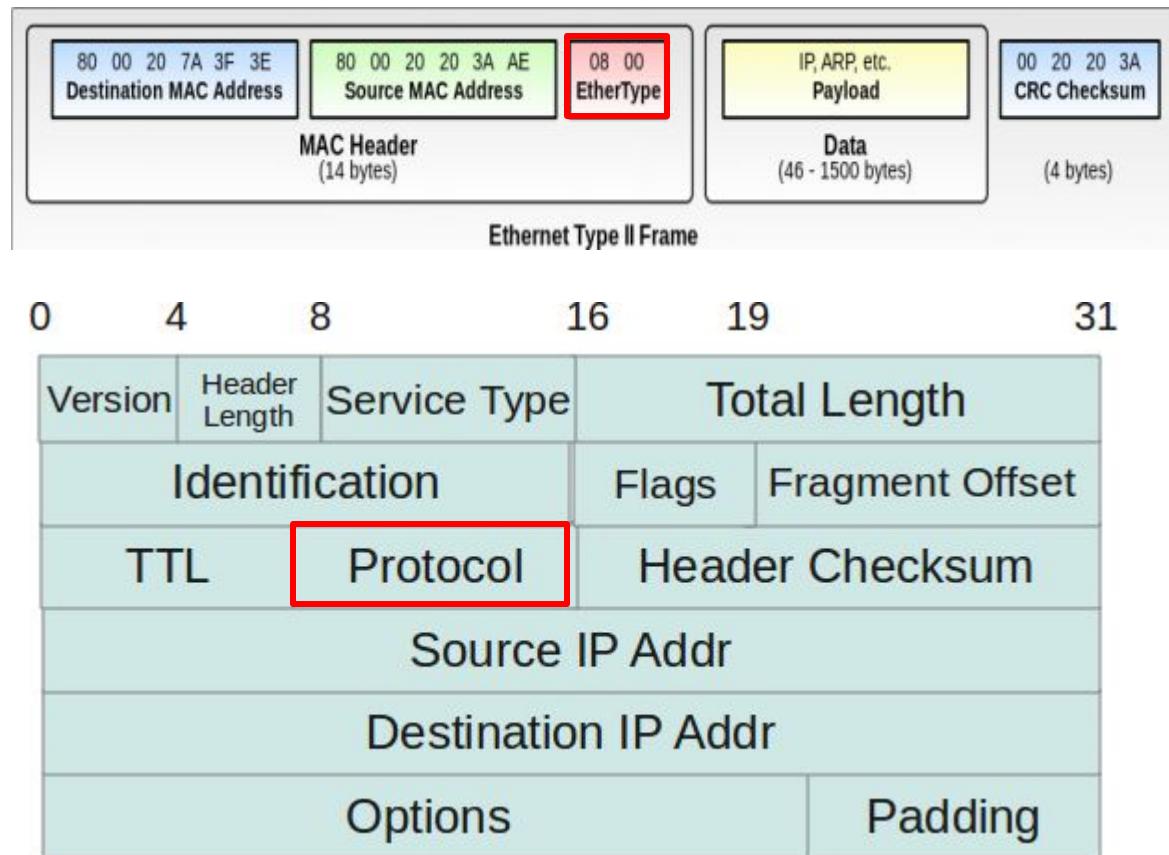


Exemplo programa BPF

- `tcpdump -i eno1 -d IPv4_TCP_packet`

- Código BPF

```
ldh [12]
jne #0x800, drop
ldb [23]
jne #6, drop
ret # -1
drop: ret #0
```



Extend Berkeley Packet Filter (eBPF)

- Várias atualizações foram realizadas no BPF;
 - BPF foi denominado como cBPF (BPF clássico);
- Em 2014, eBPF foi inserido na versão 3.15 do *kernel*;
 - Novas instruções foram adicionadas ao conjunto de instruções;
 - Aumento no número e largura dos registadores;
 - Suporte a pilha, mapas, funções auxiliares e *tail calls*;
 - Máquina virtual com carregamento dinâmico;
 - Uso da linguagem C;

Quem está usando eBPF?



Junho 2018: Balanceador de carga camada 4 (Ktran). [\[Referência1\]](#).



Fevereiro 2018: BPF vêm através de Firewall. [\[Referência1\]](#), [\[Referência2\]](#) e [\[Referência3\]](#)

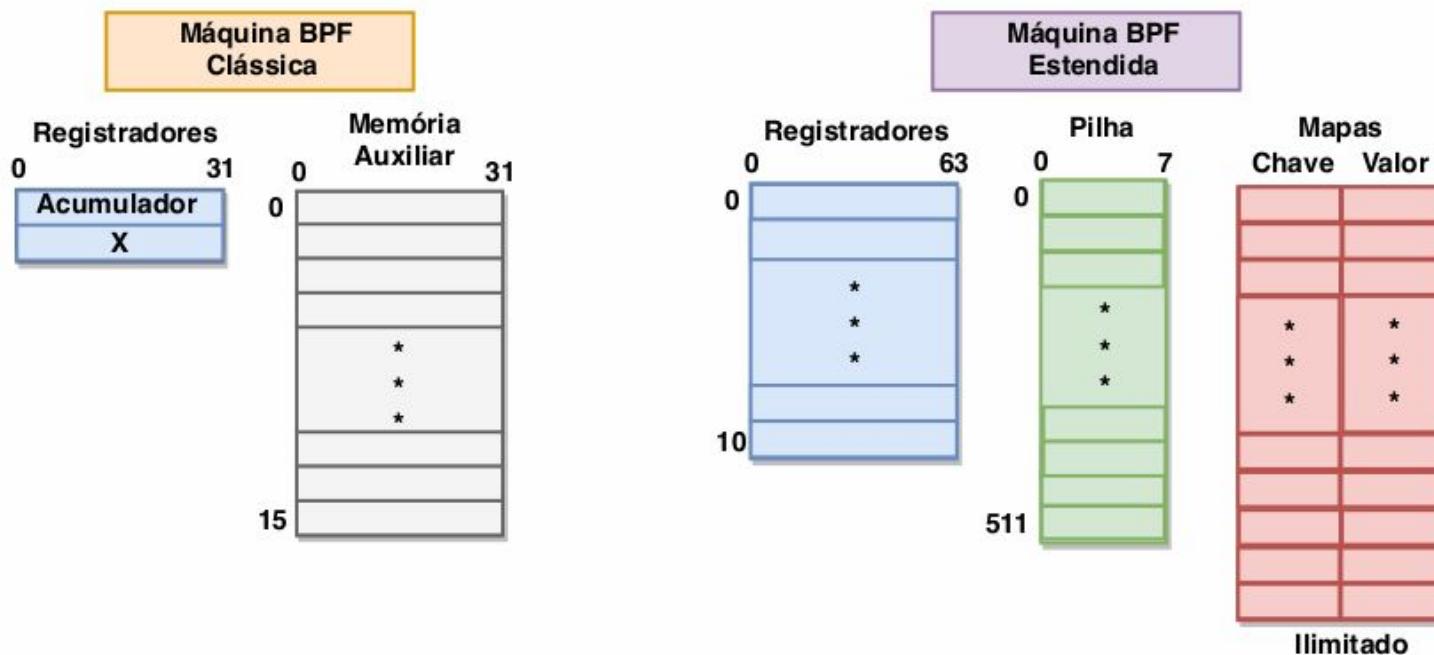


Março 2018: Introduzido suporte ao AF_XDP para trazer pacotes do driver da NIC diretamente para o espaço de usuário. [\[Referência1\]](#), [\[Referência2\]](#) e [\[Referência3\]](#).



Abril 2018: Exemplos de encaminhamento IPv4 e IPv6 inseridos no XDP para explorar a tabela de roteamento do Linux e encaminhar pacotes no eBPF. [\[Referência\]](#)

Diferenças entre BPF e eBPF



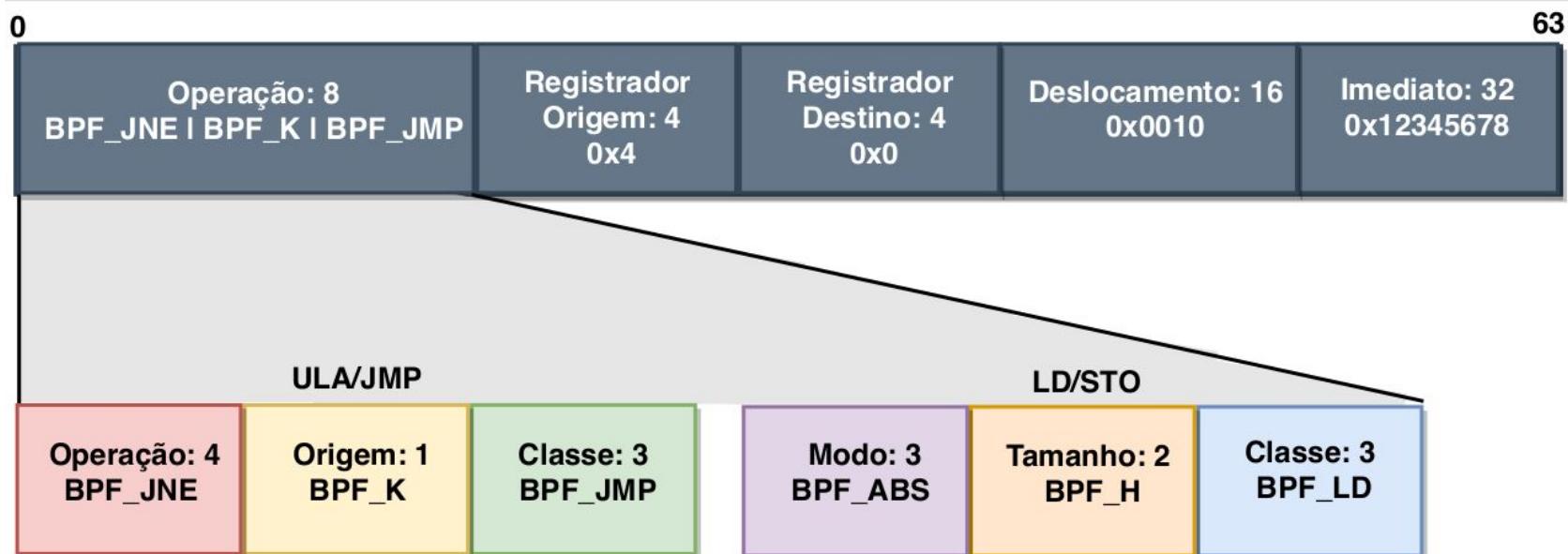
- Número de registradores: 2 para 11;
- Largura registradores: 32-bits para 64-bits;
- 11 registradores de 64 bits, pilha de 512 bytes e mapas ilimitado;
- Instruções de 64 bits.
 - 4.096 instruções por programa (máximo);

eBPF - Conjunto de instruções

- 7 classes
 - **BPF_LD, BPF_LDX**: Ambas as classes são para operações de carregar (*load*);
 - **BPF_ST, BPF_STX**: Ambas as classes são para operações de armazenar (*store*);
 - **BPF_ALU**: Operações ALU de 32 bits;
 - **BPF_ALU64**: Operações ALU de 64 bits;
 - **BPF_JMP**: Esta classe é dedicada às operações de salto (*jump*). Saltos podem ser não-condicionais e condicionais;
- Leitura/Escrita palavras: byte (8 bits); meia palavra (16 bits); palavra (32 bits); palavra dupla (64 bits).

eBPF - Bytecode

- Instruções de 64 bits;
 - Código da operação (8 bits);
 - 2 operandos origem e destino (4 bits cada);
 - Deslocamento (16 bits);
 - Imediato (32 bits).



eBPF - Registradores

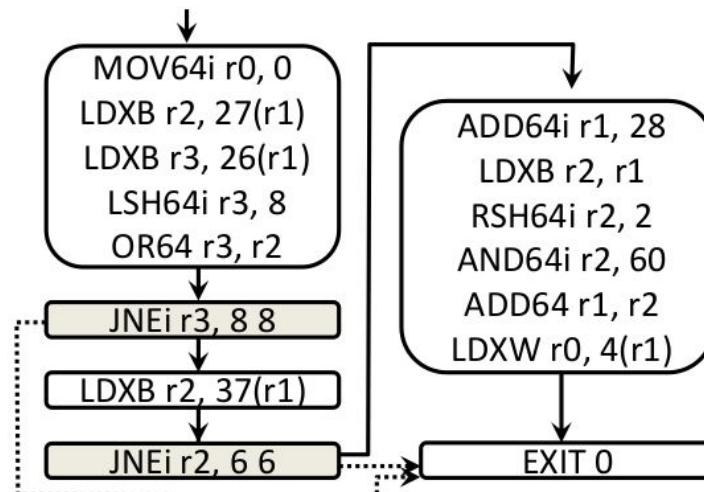
- **R0:** Valor de retorno de funções e da saída do programa eBPF;
- **R1 - R5:** Argumentos função programa eBPF;
- **R6 - R9:** Registradores que preservam os valores em chamadas de função.
- **R10:** Ponteiro do quadro para acesso a pilha. É apenas de leitura.

eBPF - Verificador

- **Segurança e estabilidade do *kernel***
 - Código eBPF injetado dentro do *kernel* precisa ser seguro;
 - Riscos potenciais
 - Laço de repetição pode travar o *kernel*;
 - *Overflows* no buffer;
 - Variáveis não inicializadas;
 - Programas grandes podem causar problemas de desempenho.

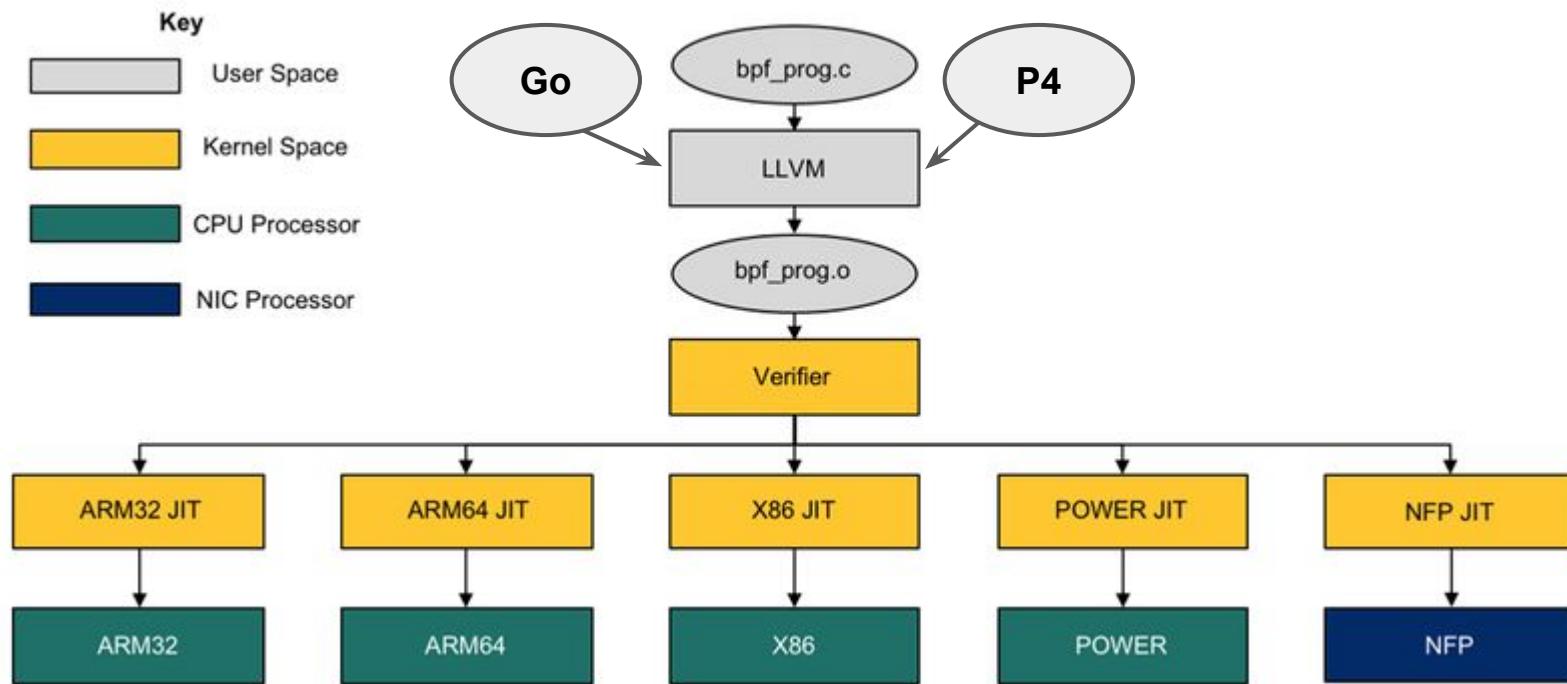
eBPF - Verificador

- Permite checar a validade, segurança e desempenho de programas;
- Funcionalidades:
 - Verifica estaticamente se um programa termina;
 - Verifica se os endereços de acesso a memória são válidos;
 - Verifica se existe laços de repetição;
 - ...
- Uso de grafo acíclico direcionado (GAD)



eBPF - Linguagens de alto nível

- Linguagens C, Go e P4 podem ser compiladas para eBPF.



Baseada em: [Blog Netronome](#)

eBPF - Linguagens de alto nível

- Restrições C para eBPF
 - Subconjunto de C (ex: sem printf());
 - Sem variáveis globais (uso de mapas);
 - Sem laços (uso de diretivas pragma unroll);
 - Máximo 4.096 instruções por programa;
 - Espaço da pilha limitado até 512 bytes.

eBPF - Linguagens de alto nível

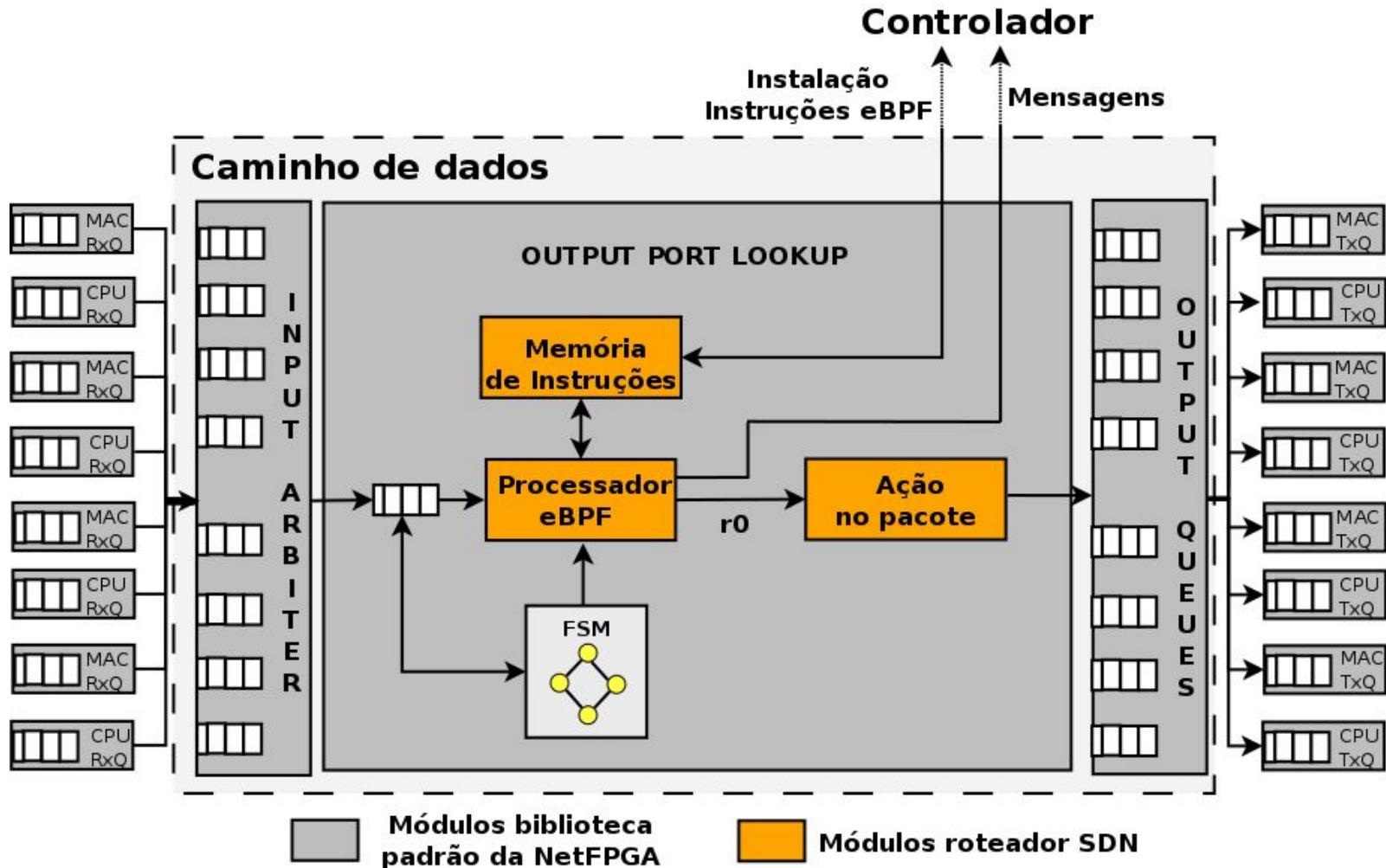
- Restrições P4
 - P4-14 tem algumas restrições essenciais;
 - Instruções IF-ELSE podem apenas serem usadas no bloco de controle;
 - Não suporta instrução for (laço);
 - Tem um conjunto limitado de ações primitivas.
 - P4 para eBPF: [Github¹](#)/[Github²](#).

eBPF - Plataformas

Hardware	Software
SmartNIC Netronome	<i>Kernel / uBPF</i>
<u>Roteador SDN</u>	<u>BPFabric</u>

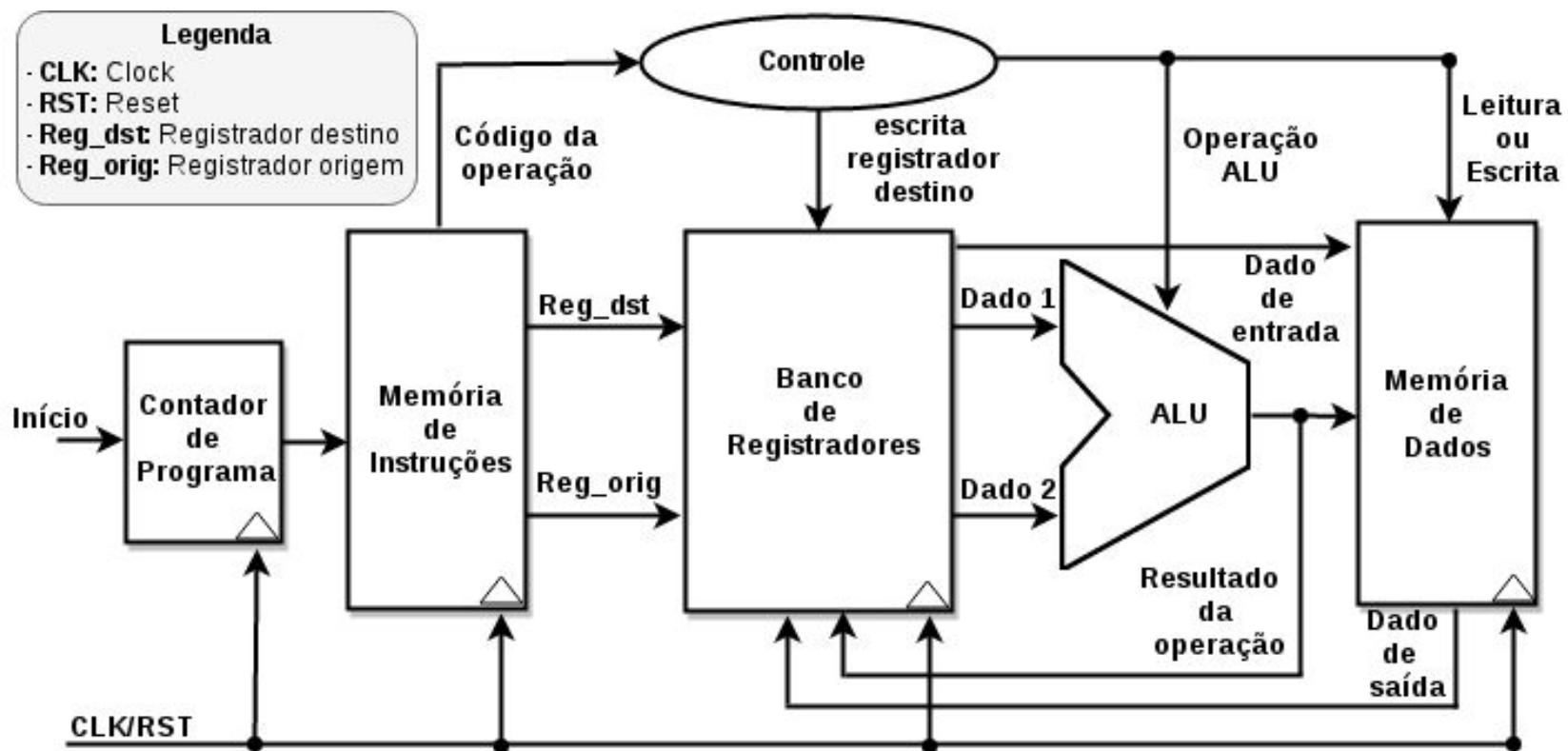
Roteador SDN

- Caminho de dados



Roteador SDN

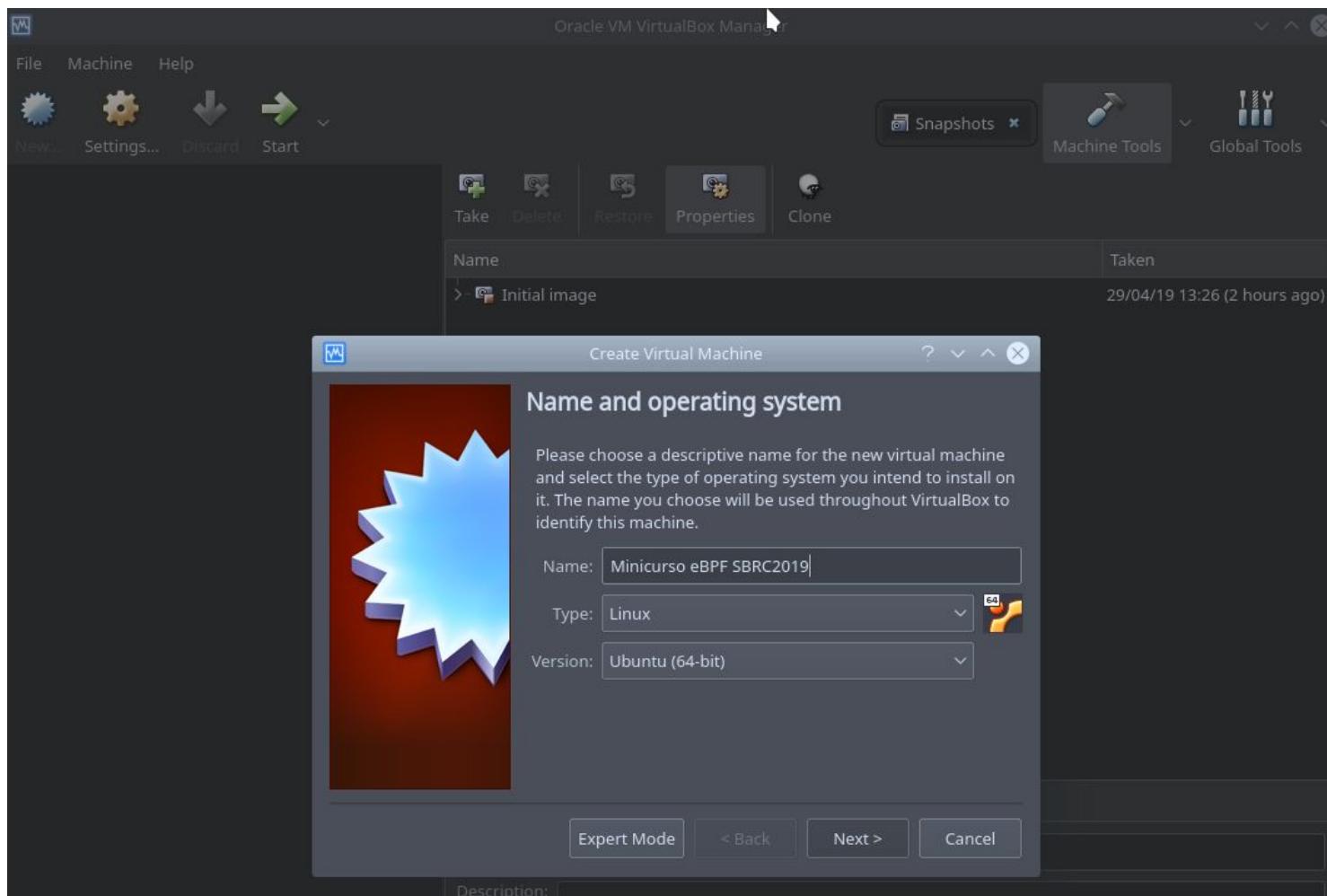
- Processador eBPF



Tipos de ganchos e primeiros passos

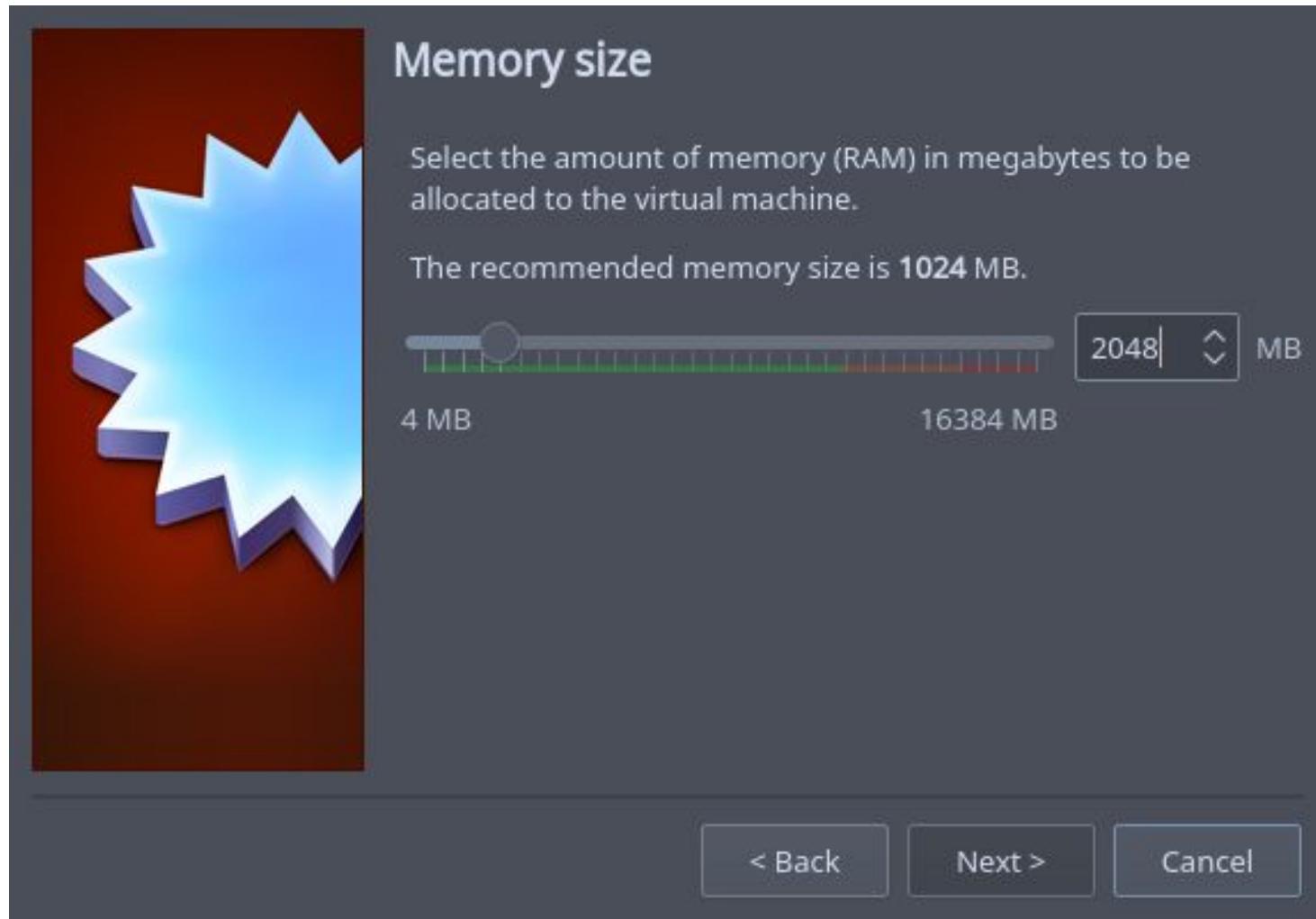
Configuração da máquina virtual

- Criar nova VM



Configuração da máquina virtual

- Selecionar quantidade de RAM



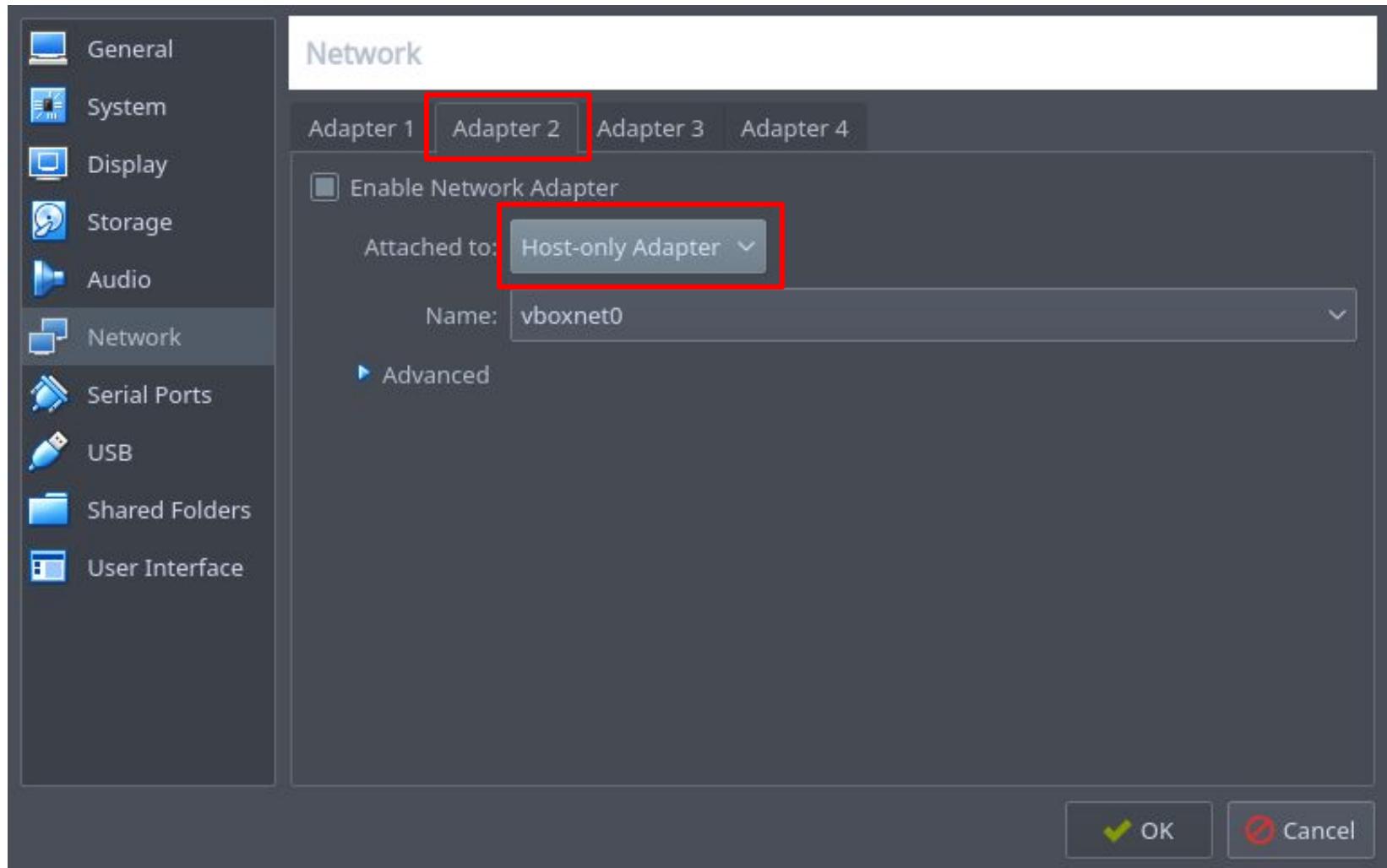
Configuração da máquina virtual

- Selecionar imagem do minicurso como disco



Configuração da máquina virtual

- Após a criação, adicionar interface Host-only

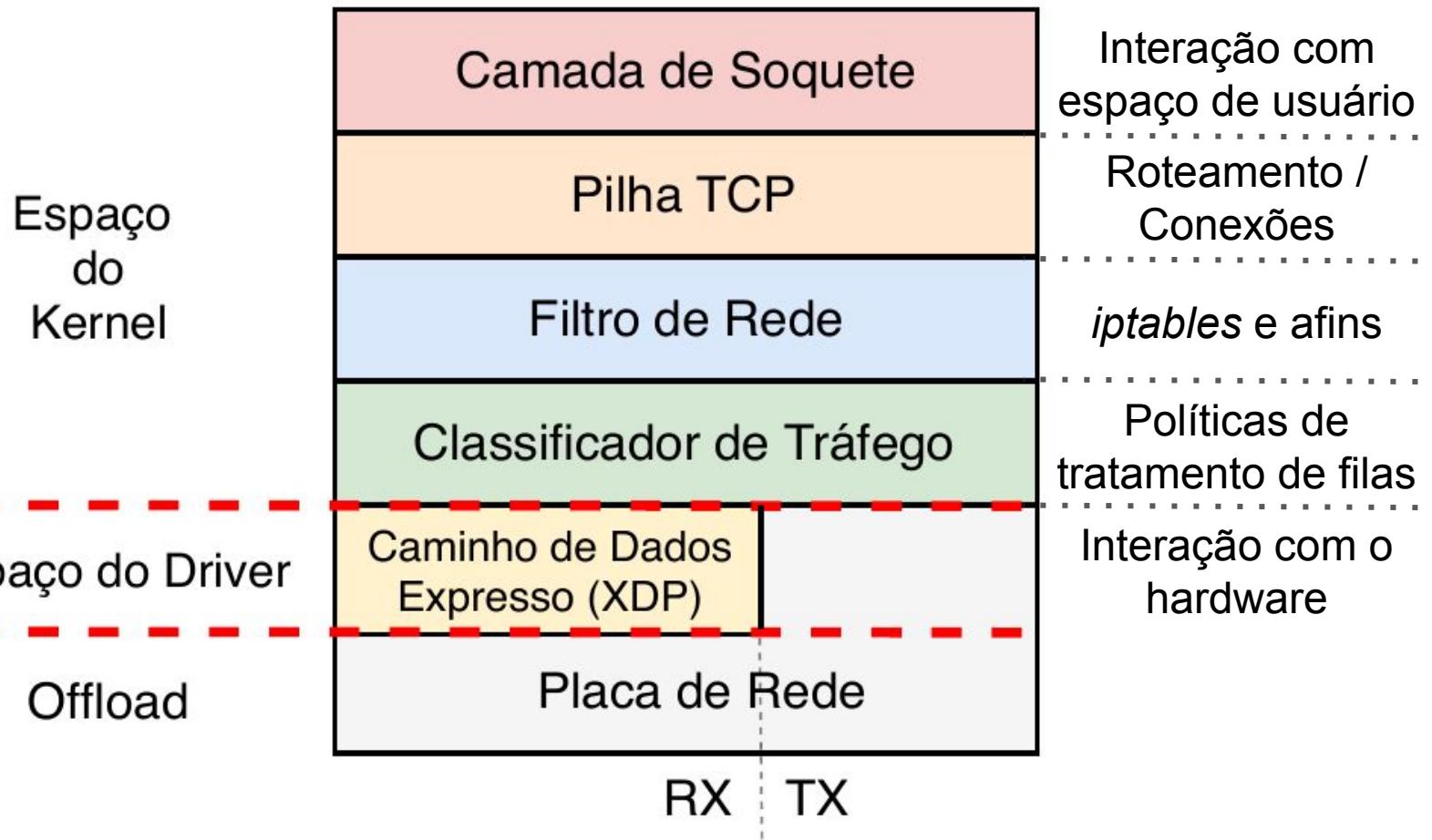


Ganchos (Hooks)

- “callback” no *kernel*;
- Código que manipula chamadas de função interceptadas, eventos ou mensagens entre componentes de software;
- Permite customizar o processamento de pacotes feito pela pilha de rede do *kernel*.

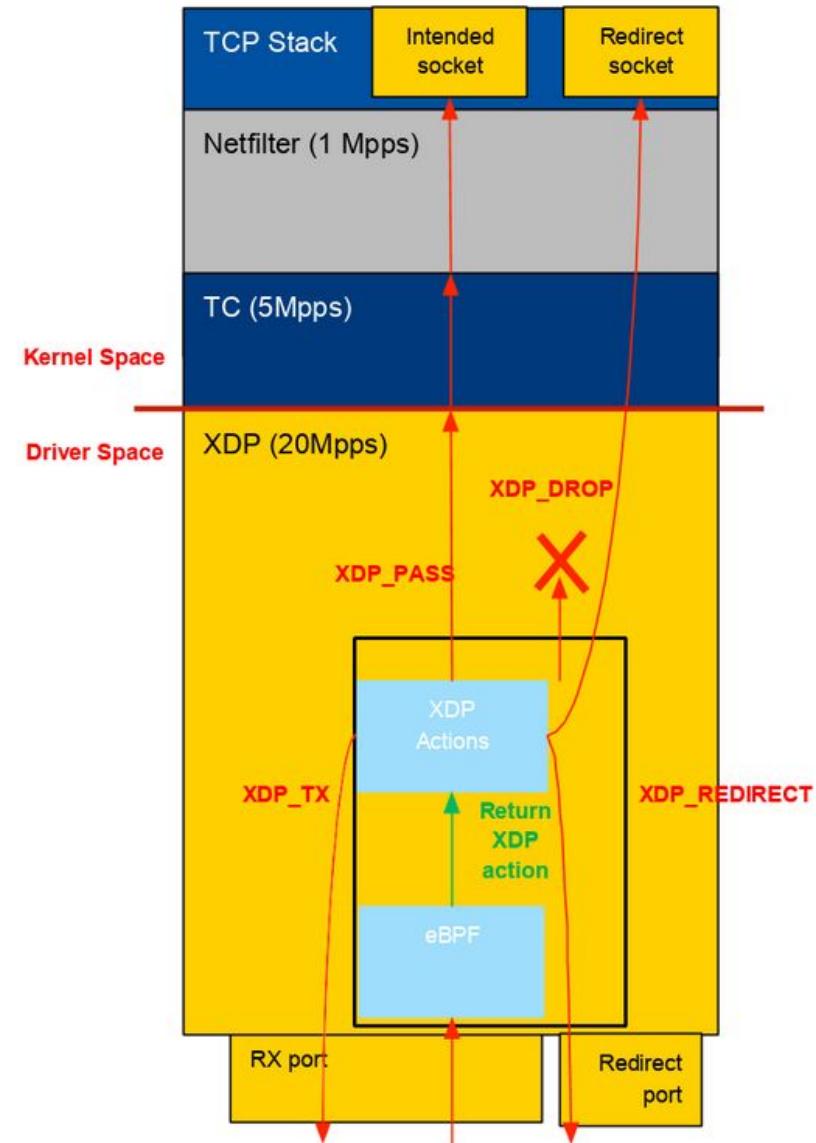


Camadas do *kernel*



Gancho express Data Path (XDP)

- Gancho no ponto mais baixo da pilha de rede do *kernel*;
- Em geral implementado no driver da placa de rede;
- Opera antes da criação de estruturas internas do *kernel* (*socket buffers*);
- Maior desempenho;
- **Presente apenas no RX.**



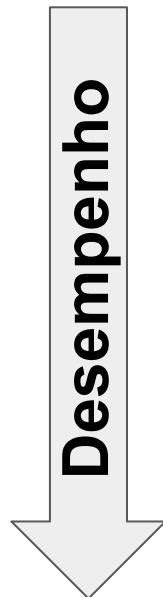
Fonte: [Tutorial eBPF SIGCOMM 2018](#)

Ações XDP

- As ações possíveis para o gancho XDP são:
 - **XDP_ABORTED**: Erro. Descarta o pacote;
 - **XDP_DROP**: Descarta o pacote;
 - **XDP_PASS**: Permite o pacote continuar até o *kernel*;
 - **XDP_TX**: Devolve o pacote para a rede;
 - **XDP_REDIRECT**: Redireciona o pacote para outra interface.
- Valores definidos em *linux/include/uapi/linux/bpf.h*

Modos de operação XDP

- Programas XDP podem rodar em 3 modos:
 - **XDP Generic**: gancho “emulado” pelo *kernel*. Permite uso com drivers sem suporte nativo;
 - **XDP Native**: gancho no driver da placa de rede. Requer suporte do driver¹;
 - **XDP Offload**: código executado em hardware (ex: SmartNIC).



¹ Lista de drivers com suporte ao XDP nativo: [BCC Docs: kernel-versions.md](#)

Primeiros passos: *Drop World!*

- Primeiro programa eBPF:

```
#include <linux/bpf.h>
int prog(struct xdp_md *ctx) {
    return XDP_DROP;
}
```

- Compilando:

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

- Carregando:

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

eBPF ELF

- Dump com *llvm-objdump*:

```
$ llvm-objdump -S dropworld.o
```

```
ebpf@osboxes:~/bpf-tutorial/examples$ llvm-objdump -S dropworld.o

dropworld.o:      file format ELF64-BPF

Disassembly of section .text:
prog:
    0:      b7 00 00 00 02 00 00 00          r0 = 2
    1:      95 00 00 00 00 00 00 00          exit
```

- **.text** é a seção ELF padrão do código compilado

Status do programa

- Conferindo status do programa:

```
$ ip link show [DEV]
```

```
ebpf@osboxes:~/Documents$ ip link show eth1
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdpgeneric
    qdisc fq_codel state UP mode DEFAULT group default qlen 1000
        link/ether 08:00:27:f7:ae:b9 brd ff:ff:ff:ff:ff:ff
        prog/xdp id 14
```

- Removendo o programa:

```
# ip link set dev [DEV] xdp off
```

```
ebpf@osboxes:~/Documents$ sudo ip link set dev eth1 xdp off
ebpf@osboxes:~/Documents$ ip link show dev eth1
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel
    state UP mode DEFAULT group default qlen 1000
        link/ether 08:00:27:f7:ae:b9 brd ff:ff:ff:ff:ff:ff
```

Offloading para SmartNIC

- A escolha entre *XDP Generic* e *XDP Native* é feita automaticamente pelo *kernel* com base no driver da interface;
- *Offloading* requer indicação explícita, além de uma placa de rede compatível (SmartNIC);
- Carregamento:

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

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

dropworld.c em detalhes

A função pode ter qualquer nome, o que indica o início do programa é a seção no arquivo ELF

Biblioteca contendo os *structs* e *enums* utilizados pelo eBPF

```
#include <linux/bpf.h>
int prog(struct xdp_md *ctx) {
    return XDP_DROP;
}
```

Tipo de retorno é o valor de um enum declarado em *bpf.h*. Depende do gancho.

Contexto passado a um programa XDP, contém ponteiros para o pacote original + metadados. Depende do gancho.

Seções ELF

- O programa eBPF a ser carregado é indicado pela seção do arquivo ELF gerado após a compilação;
 - **.text** é a seção ELF padrão do código compilado;
 - É possível especificar o nome da seção, e com isso declarar múltiplos programas eBPF no mesmo arquivo .c
- ```
#include <linux/bpf.h>

#include "bpf_helpers.h"

SEC("drop-all")
int drop(struct xdp_md *ctx) {
 return XDP_DROP;
}

SEC("pass-all")
int pass(struct xdp_md *ctx) {
 return XDP_PASS;
}
```

# Contexto XDP

- Extraído de linux/include/uapi/linux/bpf.h

```
/* user accessible metadata for XDP packet hook
 * new fields must be added to the end of this structure
 */
espaço livre para meta dados →
struct xdp_md {
 __u32 data; ← início do contexto
 __u32 data_end; ← fim do pacote
 __u32 data_meta;
 /* Below access go through struct xdp_rxq_info */
 __u32 ingress_ifindex; /* rxq->dev->ifindex */
 __u32 rx_queue_index; /* rxq->queue_index */
};
```

Exemplo de uso de metadados: linux/samples/bpf/xdp2skb\_meta\_kern.c

# Acesso ao pacote

---

- Acesso feito utilizando os ponteiros fornecidos pelo contexto:

```
void *data_end = (void *) (long) ctx->data_end;
void *data = (void *) (long) ctx->data;
```

- A conversão acima é padrão, de forma que essas duas linhas podem ser copiadas no início de todo programa eBPF;
- No XDP, *ctx->data* aponta para o início do quadro Ethernet.

# exemplos/portfilter.c

----- | início do arquivo omitido

```
SEC("portfilter")
int pfilter(struct xdp_md *ctx) {

 void *data_end = (void *)(long)ctx->data_end;
 void *data = (void *)(long)ctx->data;
 struct ethhdr *eth = data;
 __u32 eth_proto;
 __u32 nh_off;

 nh_off = sizeof(struct ethhdr);
 if (data + nh_off > data_end)
 return XDP_PASS;
 eth_proto = eth->h_proto;

 /* demo program only accepts ipv4 packets */
 if (eth_proto == bpf_htons(ETH_P_IP))
 return process_packet(ctx, nh_off);
 else
 return XDP_PASS;
}
```

# exemplos/portfilter.c

----- | início do arquivo omitido

```
SEC("portfilter")
int pfilter(struct xdp_md *ctx) {

 void *data_end = (void *) (long) ctx->data_end;
 void *data = (void *) (long) ctx->data;
 struct ethhdr *eth = data;
 __u32 eth_proto;
 __u32 nh_off;

 nh_off = sizeof(struct ethhdr);
 if (data + nh_off > data_end)
 return XDP_PASS;
 eth_proto = eth->h_proto;

 /* demo program only accepts ipv4 packets */
 if (eth_proto == bpf_htons(ETH_P_IP))
 return process_packet(ctx, nh_off);
 else
 return XDP_PASS;
}
```

Checagem de limites  
obrigatória. Se não for  
feita, o código é  
rejeitado pelo verificador

# Lidando com o verificador

----- | início do arquivo omitido

```
SEC("portfilter")
int pfilter(struct xdp_md *ctx) {

 void *data_end = (void *) (long) ctx->data_end;
 void *data = (void *) (long) ctx->data;
 struct ethhdr *eth = data;
 __u32 eth_proto;
 __u32 nh_off;

 nh_off = sizeof(struct ethhdr);
 if (data + nh_off > data_end)
 return XDP_PASS;
 eth_proto = eth->h_proto;

 /* demo program only accepts ipv4 packets */
 if (eth_proto == bpf_htons(ETH_P_IP))
 return process_packet(ctx, nh_off);
 else
 return XDP_PASS;
}
```

Vamos retirar esse trecho de código e tentar carregar o programa no *kernel*

# Lidando com o verificador

```
ebpf@osboxes:~/bpf-tutorial/exemplos$ sudo ip -force link set
dev eth0 xdp obj select.o sec xdp

Prog section 'xdp' rejected: Permission denied (13)!
- Type: 6
- Instructions: 22 (0 over limit)
- License:

Verifier analysis:

0: (61) r2 = *(u32 *)(r1 +0)
1: (71) r3 = *(u8 *)(r2 +13)
invalid access to packet, off=13 size=1, R2(id=0,off=0,r=0)
R2 offset is outside of the packet

Error fetching program/map!
```

# Mapas

---

- eBPF suporta o uso de mapas, que são estruturas de armazenamento do tipo chave-valor;
- Até a versão 5.0 do *kernel*, existem 23 tipos de mapas. Alguns deles são:
  - BPF\_MAP\_TYPE\_HASH
  - BPF\_MAP\_TYPE\_ARRAY
  - BPF\_MAP\_TYPE\_LPM\_TRIE
  - BPF\_MAP\_TYPE\_PROG\_ARRAY
  - ...
- Tipos de mapas definidos na biblioteca *bpf.h*

# Mapas (2)

---

- A declaração de um mapa é feita como uma variável global no código em C:

```
struct bpf_map_def SEC("maps") rxcnt = {
 .type = BPF_MAP_TYPE_PERCPU_ARRAY,
 .key_size = sizeof(u32),
 .value_size = sizeof(long),
 .max_entries = 256,
};
```

- Declaração da seção *maps* é obrigatória para geração correta do arquivo ELF pelo compilador.

# Funções auxiliares

---

- O *kernel* oferece diversas funções auxiliares para serem usadas por programas eBPF;
- Overhead de chamadas para funções auxiliares é quase zero;
- Arquivos *.h* localizados em */linux/tools/testing/selftests/bpf/*
  - ***bpf\_helpers.h*** : funções diversas;
  - ***bpf\_endian.h*** : conversão de extremidade (*endianness*);
- Ambos podem ser copiados localmente para facilitar a compilação

# Funções auxiliares (2)

---

- Algumas das funções disponíveis:
  - **bpf\_map\_lookup\_elem**, **bpf\_map\_update\_elem**, **bpf\_map\_delete\_elem**: interação com mapas
  - **bpf\_ntohs**, **bpf\_htons**: conversão de extremidade
  - **bpf\_get\_prandom\_u32**: retorna um valor de 32 bits pseudo-aleatório.
  - **bpf\_redirect**: redireciona pacote para outra interface
  - **bpf\_getsockopt** e **bpf\_setsockopt**: configuração de soquetes
  - **bpf\_tail\_call**: inicia a execução de outro programa eBPF

# Funções auxiliares (3)

- As funções auxiliares disponíveis variam conforme o gancho utilizado
- Funções disponíveis no XDP:

```
BPF_FUNC_map_lookup_elem()
BPF_FUNC_map_update_elem()
BPF_FUNC_map_delete_elem()
BPF_FUNC_map_peek_elem()
BPF_FUNC_map_pop_elem()
BPF_FUNC_map_push_elem()
BPF_FUNC_get_prandom_u32()
BPF_FUNC_get_smp_processor_id()
BPF_FUNC_get numa_node_id()
BPF_FUNC_tail_call()
BPF_FUNC_ktime_get_ns()
BPF_FUNC_trace_printk()
BPF_FUNC_spin_lock()
BPF_FUNC_spin_unlock()
```

```
BPF_FUNC_perf_event_output()
BPF_FUNC_get_smp_processor_id()
BPF_FUNC_csum_diff()
BPF_FUNC_xdp_adjust_head()
BPF_FUNC_xdp_adjust_meta()
BPF_FUNC_redirect()
BPF_FUNC_redirect_map()
BPF_FUNC_xdp_adjust_tail()
BPF_FUNC_fib_lookup()
```

Fonte: [Projeto BCC](#)

# Funções auxiliares (3)

- As funções auxiliares disponíveis variam conforme o gancho utilizado
- Funções disponíveis no XDP:

BPF\_FUNC\_map\_lookup\_elem()  
BPF\_FUNC\_map\_update\_elem()  
BPF\_FUNC\_map\_delete\_elem()  
BPF\_FUNC\_map\_peek\_elem()  
BPF\_FUNC\_map\_pop\_elem()  
BPF\_FUNC\_map\_push\_elem()  
BPF\_FUNC\_get\_prandom\_u32()  
BPF\_FUNC\_get\_smp\_processor\_id()  
BPF\_FUNC\_get numa\_node\_id()  
BPF\_FUNC\_tail\_call()  
BPF\_FUNC\_ktime\_get\_ns()  
BPF\_FUNC\_trace\_printk()  
BPF\_FUNC\_spin\_lock()  
BPF\_FUNC\_spin\_unlock()

BPF\_FUNC\_perf\_event\_output()  
BPF\_FUNC\_get\_smp\_processor\_id()  
BPF\_FUNC\_csum\_diff()  
BPF\_FUNC\_xdp\_adjust\_head()  
BPF\_FUNC\_xdp\_adjust\_meta()  
BPF\_FUNC\_redirect()  
BPF\_FUNC\_redirect\_map()  
BPF\_FUNC\_xdp\_adjust\_tail()  
BPF\_FUNC\_fib\_lookup()

Fonte: [Projeto BCC](#)

# Interagindo com mapas

```
void *bpf_map_lookup_elem(struct bpf_map *map,
 const void *key)
```

```
int bpf_map_delete_elem(struct bpf_map *map,
 const void *key)
```

```
int bpf_map_update_elem(struct bpf_map *map,
 const void *key, const void *value, u64 flags)
```

*flags* pode receber  
os seguintes  
valores:

- BPF\_ANY
- BPF\_NOEXIST
- BPF\_EXIST

# Exemplo

---

**linux/samples/bpf/xdp1\_kern.c**

# Intervalo

# bpftool

---

- Ferramenta do *kernel*: *linux/tools/bpf/bpftool*;
- Capaz de ler / escrever em mapas eBPF;
- Importante para depuração.

```
root@sbrc2019:/home/ebpf# bpftool map dump id 20
key:
00 00 00 00
value (CPU 00): 05 00 00 00 00 00 00 00 00 00
key:
01 00 00 00
value (CPU 00): 03 00 00 00 00 00 00 00 00 00
key:
02 00 00 00
```

# Espaço de usuário

---

- Exemplos do *kernel* divididos em 2 partes: **\*\_user.c**  
**\*\_kern.c**
- Interação com programas eBPF a partir do espaço de usuário pode ser feita a partir da *syscall bpf()*;

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

- Mais detalhes: \$ man bpf
- Nova biblioteca ***linux/tools/lib/bpf/libbpf.h*** contém funções para auxiliar essa interação.

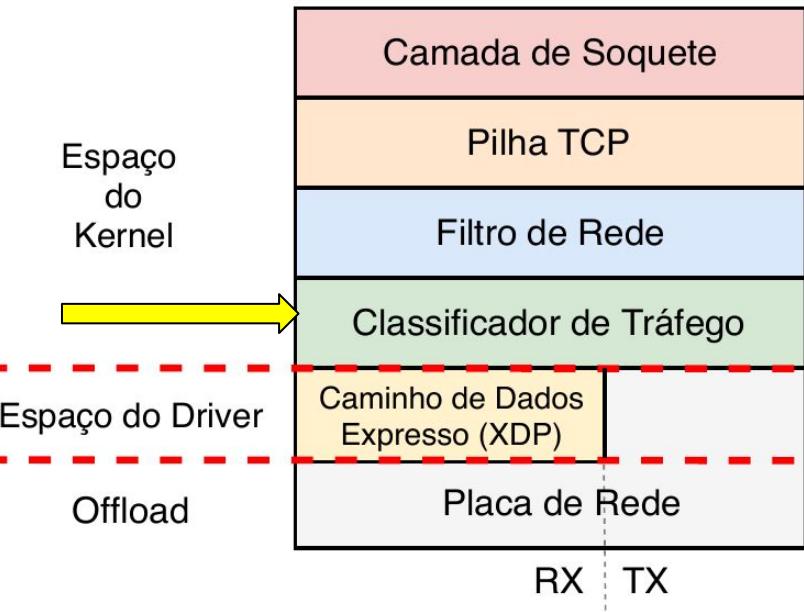
# Exemplo

---

**linux/samples/bpf/xdp1\_user.c**

# Gancho Traffic Control (TC)

- Políticas de controle de tráfego no *kernel*;
- Filtros especiais como programas eBPF;
- Tanto *ingress* quanto *egress*,
- Ações disponíveis:



| 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.   |

# Gancho Traffic Control (TC) (2)

---

- Criando a qdisc clsact:

```
tc qdisc add dev <iface> clsact
```

- Adicionando o programa:

```
tc filter add dev <iface> <direction> bpf da obj \
 <ebpf-obj> sec <section>
```

*<direction>* = *ingress* ou *egress*

- Checando status:

```
tc filter show dev <iface> <direction>
```

# Exemplo

---

**linux/samples/bpf/xdp2skb\_meta\_kern.c**

# Outros conceitos

---

- Tipos de programas;
- Chamadas de cauda (*tail calls*);
- Soquetes AF\_XDP;
- eBPF em outras camadas do *kernel*;
- Introspecção do *kernel* com eBPF (ver referências);
- Compiladores JIT;
- Map pinning;
- ...

# **Expandindo eBPF com BPFabric**

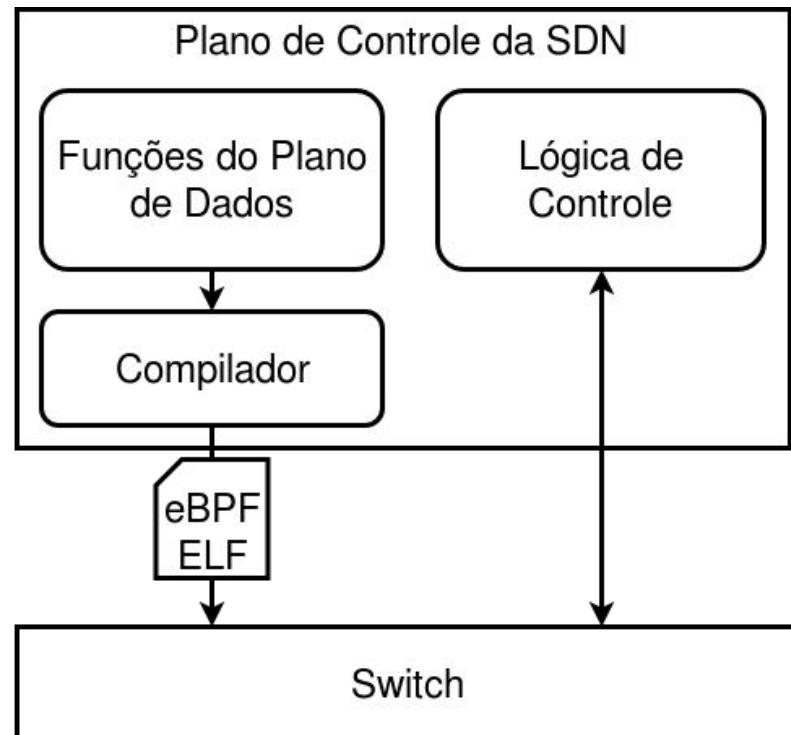
# Introdução ao BPFabric

---

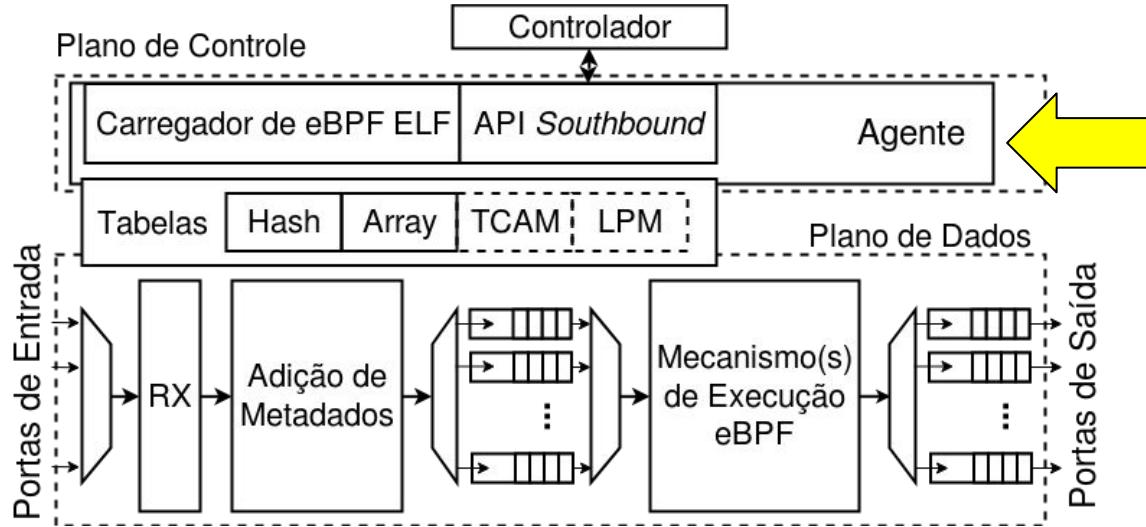
- OpenFlow não é *future-proof*
  - Suporte a novos protocolos ou ações requer novas especificações.
- BPFabric é uma arquitetura SDN independente de
  - Plataforma;
  - Protocolo;
  - Linguagem.
- Switch também é “burro”;
- eBPF como conjunto de instruções para as funções do plano de dados;
- Permite a adição dinâmica de novas funções ao switch.

# Visão Geral do Controlador no BPFabric

- Funções para o switch são desenvolvidas em linguagem de alto nível
  - Ex.: C (restrito).
- Compilador traduz funções da linguagem para eBPF
  - Gera eBPF ELF.
- Lógica de controle lida com
  - Envio de pacotes;
  - Notificação de eventos;
  - Dados das tabelas.

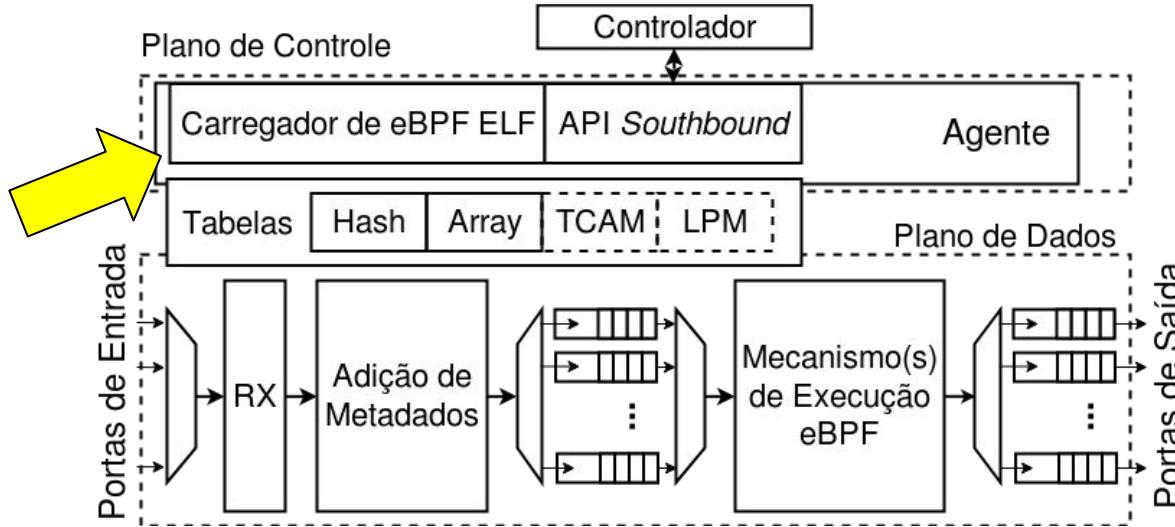


# Visão Geral do Switch no BPFabric



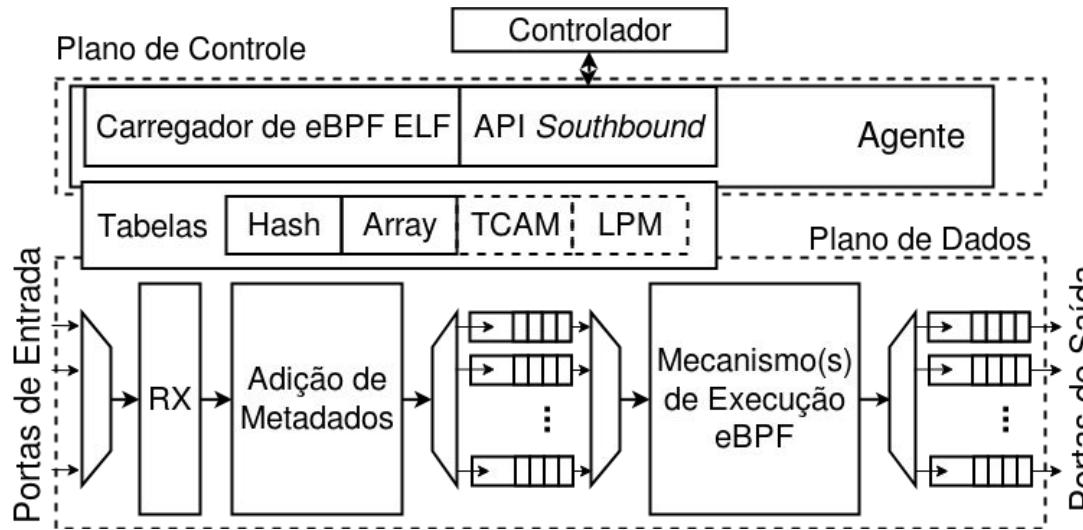
- Agente é responsável por
  - Alterar o comportamento do switch;
  - Receber pacotes e notificar eventos;
  - Ler e alterar entradas das tabelas (mapas).
- Comunicação feita através da API Southbound.

# Visão Geral do Switch no BPFabric (2)



- Carregador de eBPF ELF é responsável por
  - Alocar tabelas (mapas) eBPF;
  - Converter código de byte eBPF para formato do dispositivo.
- Específico para cada dispositivo
  - Controlador pode ser independente de plataforma.
- Deve verificar se programa é válido, seguro e rápido.

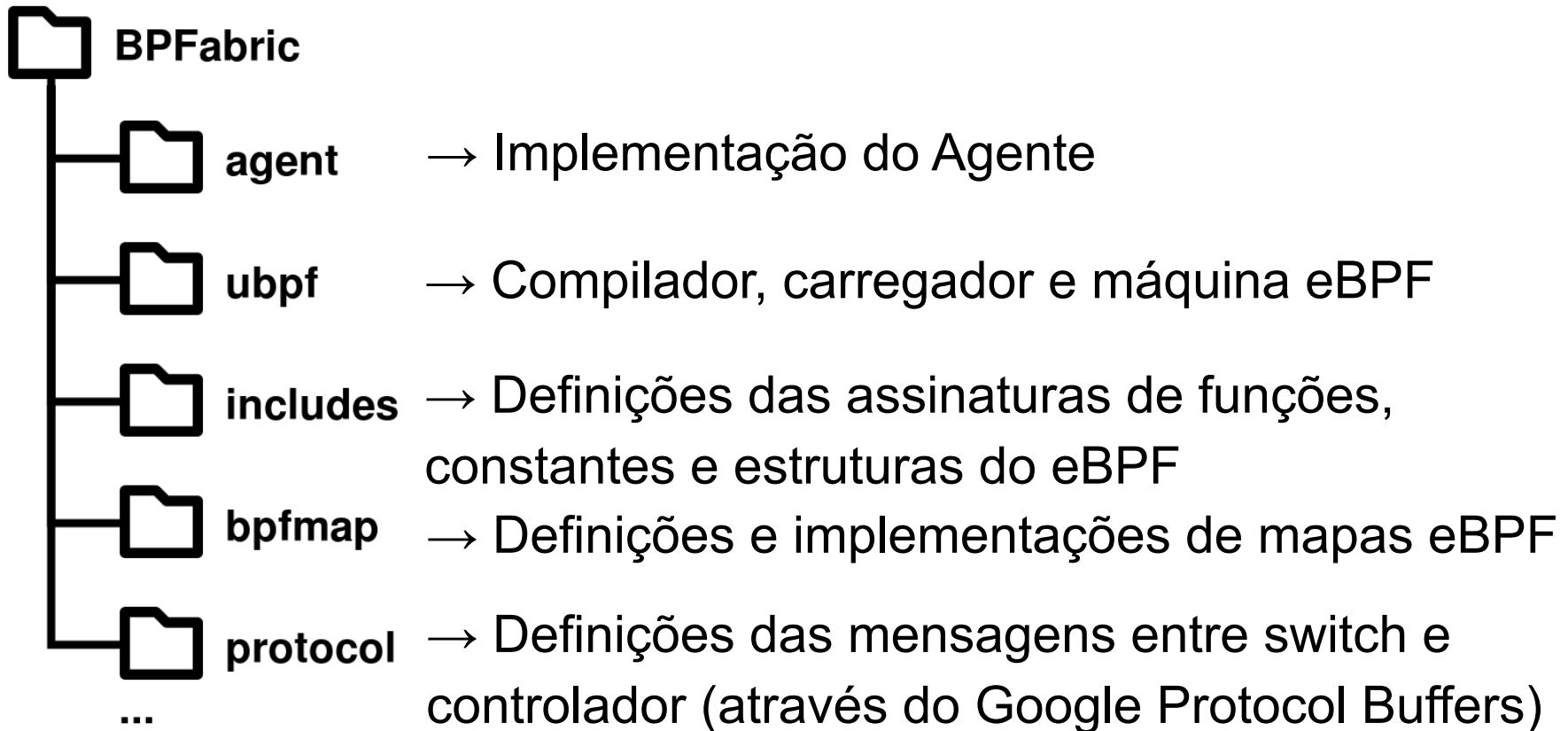
# Visão Geral do Switch no BPFabric (3)



- Plano de dados possui fluxo bem definido;
- Funções carregadas são executados por um Mecanismo de Execução eBPF
  - Entrada: pacote + metadados
  - Saída: ação (*flood*, *drop*, enviar ao controlador etc)

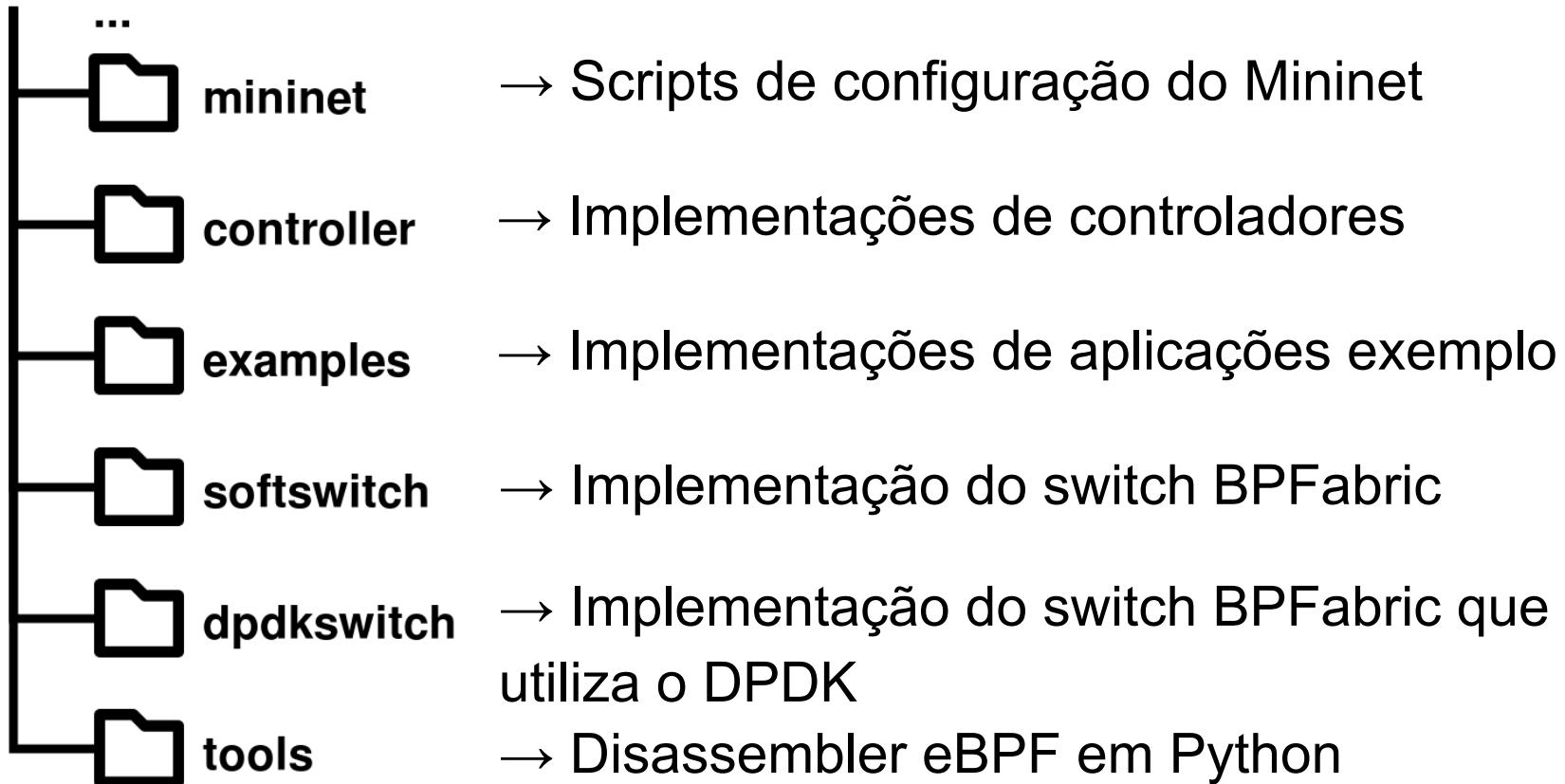
# Estrutura do BPFabric

---



# Estrutura do BPFabric (2)

---



# Exemplo

---

- ***FOO\_MAP***
  - Contador de 32 bits
  - Operações:
    - `foo_map_alloc`: Cria mapa
    - `foo_map_update_elem`: Incrementa contador
    - `foo_map_lookup_elem`: Recupera valor do contador
    - `foo_map_free`: Destroi mapa
  - ***exemplos/BPFabric***
    - Código fonte: `foo_map.c` e `foo_map.h`
    - Controlador: `foo_counter.py`
    - Aplicação: `foo_counter.c`

# **Ferramentas**

# eBPF - Ferramentas

---

- **Bpftool**
  - Lista programas eBPF e mapas ativos;
  - Interação com mapas eBPF (consulta e atualização);
  - Dump código *assembly* (JIT e Pré-JIT);
- **Iprouter2**
  - Pode carregar e anexar programas eBPF no TC, XDP ou XDP offload (*SmartNic*);
- **Libbpf**
  - Biblioteca BPF que permite acessar programas no espaço do usuário através de uma API eBPF;

# eBPF - Ferramentas

---

- **llvm-objdump**
  - Imprime o código eBPF executável (ELF) no formato legível para humanos;
- **BCC - BPF *Compiler Collection***
  - Conjunto de ferramentas para monitorar o *kernel* e manipulação de programas eBPF.

# **Funções de rede e Projetos de pesquisa**

# Funções de rede

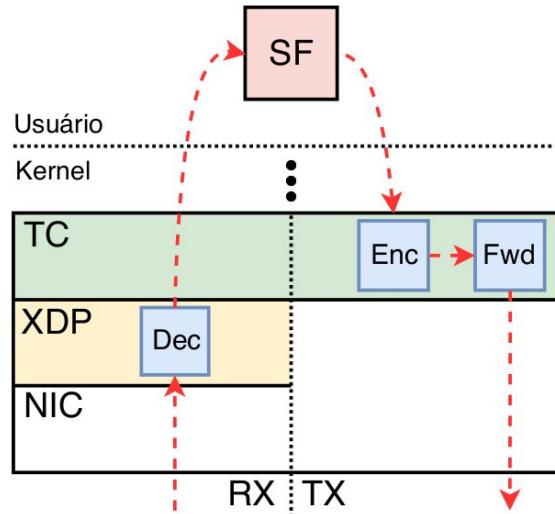
| Nome                                              | Descrição                                                                                                                                                                                                       | Plataforma            | Repositório                                                                    |
|---------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|--------------------------------------------------------------------------------|
| <b>Balanceador<br/>de<br/>carga L4<br/>(l4lb)</b> | <ul style="list-style-type: none"><li>- Hash para identificar o pacote no mapa eBPF;</li><li>- Mapa eBPF contém o endereço dos servidores disponíveis;</li><li>- Scripts em Python (<i>bpftrace</i>);</li></ul> | SmartNic<br>Netronome | <a href="#">Github</a><br><a href="#">Netronome</a><br><a href="#">l4lb</a>    |
| <b>RSS<br/>programável</b>                        | <ul style="list-style-type: none"><li>- Escalonador programável para enviar pacotes para múltiplas CPUs;</li></ul>                                                                                              | SmartNic<br>Netronome | <a href="#">Github</a><br><a href="#">Netronome</a>                            |
| <b>Monitoramento</b>                              | <ul style="list-style-type: none"><li>- Distribuição do tamanho do pacote;</li><li>- Tempo de chegada entre pacotes;</li><li>- Latência;</li><li>- EWMA;</li></ul>                                              | BPFabric              | <a href="#">Github</a><br><a href="#">BPFabric</a><br><a href="#">examples</a> |

# Funções de rede

| Nome                       | Descrição                                                                                                                                                             | Plataforma            | Repositório                                         |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------|-----------------------------------------------------|
| NAT                        | <ul style="list-style-type: none"><li>- Tabela NAT como um mapa eBPF (Tipo Hash);</li><li>- Funções eBPF lookup e update para acessar a tabela NAT;</li></ul>         | Roteador<br>SDN       | <a href="#">Github</a><br><a href="#">Minicurso</a> |
| Filtragem<br>de<br>pacotes | <ul style="list-style-type: none"><li>- Independente de protocolo;</li><li>- Análise, casamento e ações dinâmicas;</li></ul>                                          | Kernel<br>Linux       | <a href="#">Github</a><br><a href="#">Minicurso</a> |
| ChaCha                     | <ul style="list-style-type: none"><li>- Algoritmo de criptografia;</li><li>- Google: Criptografia simétrica do protocolo TLS;</li><li>- ChaCha8 e ChaCha20;</li></ul> | SmartNic<br>Netronome | <a href="#">Github</a><br><a href="#">Minicurso</a> |

# Projetos de pesquisa

- **Encadeamento de funções de rede**
  - Problema em ambientes NFV;
  - Serviços complexos precisam executar NFs em cadeia;
  - Artigo “*Cadeia-Aberta: Arquitetura para SFC em kernel usando eBPF*” - SBRC 2019
    - Nova arquitetura para desacoplar o encadeamento dos dispositivos de rede;
    - Ações de encadeamento executadas por programas eBPF no *kernel*.



Estágios de processamento da arquitetura Cadeia-Aberta

# Projetos de pesquisa

---

- **Roteamento por segmento**

- Programabilidade de redes demanda novas tecnologias;
- Ações diferentes de processamento do pacote em pontos específicos da rede;
  - Segmento;
  - Rótulos MPLS ou protocolo IPv6;
  - Dispositivos da rede habilitados na rede;
- Habilitado no *kernel 4.10* sobre o IPv6;
- Artigo “*Leveraging eBPF for programmable network functions with IPv6 segment routing*” - CoNEXT 2018;
  - eBPF para criar e especificar novos segmentos de forma genérica e flexível;

# Projetos de pesquisa

---

- **Monitoramento usando sketches**
  - Monitoramento implica na QoS;
  - Sketches;
  - BPFabric não tem sketches implementados;
  - Artigo “Aplicações de monitoramento de tráfego utilizando redes programáveis eBPF”- SBRC 2019
    - Estenderam o BPFabric adicionando sketches;
      - Bloom Filter, Count-Min, K-ary e PCSA;
    - Avaliaram aplicações de monitoramento;
      - Detecção de mudanças bruscas;
      - Detecção de *Heavy Hitters*;
      - Estimativa da distribuição do tamanho dos fluxos;
      - Contagem do tráfego.

# Outros projetos

| Projeto de pesquisa                                                     | Descrição                                                                          | Referência                                                              |
|-------------------------------------------------------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------|
| <b>[Jouet et al. 2015]<sup>1</sup><br/>[Tu et al. 2017]<sup>2</sup></b> | - Protocolo OpenFlow <sup>1</sup> e<br>OpenvSwitch com suporte eBPF <sup>2</sup> ; | <a href="#">Artigo<sup>1</sup></a> / <a href="#">Artigo<sup>2</sup></a> |
| <b>[Bertrone et al. 2018]</b>                                           | - Nova versão da ferramenta<br><i>iptables</i> usando eBPF;                        | <a href="#">Artigo</a>                                                  |
| <b>[Baidya et al. 2018]</b>                                             | - Controle do tráfego e replicação de<br>pacotes em cenários IoT usando<br>eBPF;   | <a href="#">Artigo</a>                                                  |

# Outros projetos

| Ferramentas código aberto                     | Descrição                                                                                 | Referência                                                               |
|-----------------------------------------------|-------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| <b>Cilium</b>                                 | - Segurança em redes de contêineres e aplicações com micro serviços;                      | <a href="#">Site</a>                                                     |
| <b>IO Visor</b>                               | - Conjunto de sub projetos baseados no eBPF;                                              | <a href="#">Site</a>                                                     |
| <b>BCC</b>                                    | - Conjunto de ferramentas para monitorar o <i>kernel</i> e manipulação de programas eBPF; | <a href="#">Github</a>                                                   |
| <b>uBPF</b>                                   | - Espaço do usuário para executar programas eBPF;                                         | <a href="#">Github</a>                                                   |
| <b>GoBPF</b>                                  | - Interação sistema eBPF usando linguagem Go;                                             | <a href="#">Github</a>                                                   |
| <b>Ply<sup>1</sup> e BPFtrace<sup>2</sup></b> | - Ferramentas para introspecção do <i>kernel</i> ;                                        | <a href="#">Github<sup>1</sup></a><br><a href="#">Github<sup>2</sup></a> |

# **Desafios e limitações eBPF**

# Desafios e limitações eBPF

---

- **Número de instruções limitado**
  - 4.096 instruções;
  - Dividir um programa (> 4.096 instruções) em subprogramas;
  - Saltar entre subprogramas (*tail calls*);
  - 8 chamadas sequentes sem *overhead*;
- **Laço de repetição**
  - Verificador detecta a presença de laços no código;
  - Diretivas “*pragma unroll*”
    - Aumenta o número de instruções;
    - Problema ao encontrar a constante de término do laço;

# Desafios e limitações eBPF

---

- **Enviar o mesmo pacote para múltiplas portas**
  - Aplicações: ARP, multicast e inundação;
  - Todas interfaces precisam ser percorridas para encaminhar o pacote;
  - Pacote precisa ser clonado em uma interface adicional;
  - Aplicação em cadeia virtual falha.
- **Processamento de pacote dirigido a evento**
  - Um pacote precisa transitar em um gancho selecionado;
  - O plano de dados reage a um único evento;
  - Módulo caminho lento (plano de controle).
- **Plano de controle simples e primitivo.**

# Conclusão

---

- Relativamente fácil de usar e tecnologia muito poderosa;
- Grandes empresas utilizando;
- Vários tópicos para serem pesquisados
  - Monitoramento, segurança, espaço de usuário, ....
- Necessidade de aumentar a comunidade
  - Usuários;
  - Desenvolvedores de aplicações fora do *kernel*;
  - Documentação
- P4 e eBPF não são necessariamente concorrentes;
- **Futuro eBPF e XDP na área.**
- **Agradecimentos**
  - Ao laboratório Winet e o SBRC pelo apoio financeiro.

# Processamento Rápido de Pacotes com eBPF e XDP

Dúvidas?  
Obrigado pela atenção!

**Alunos<sup>1</sup>:** Racyus Delano, Matheus Castanho,  
Eduardo Câmara e Elerson Santos.

**Professores<sup>2</sup>:** Marcos A. M. Vieira e Luiz F. M. Vieira

**Emails<sup>1</sup>:** {racyus,matheus.castanho,epmcj,elerson}@dcc.ufmg.br  
**Emails<sup>2</sup>:** {mmvieira, lfvieira}@dcc.ufmg.br

# **Material extra**

# ***Loop unrolling***

---

- Para garantir que o programa vai sempre terminar, o verificador não permite *loops*.
- Solução parcial: *loop unrolling*

```
#pragma clang loop unroll(full)
for(int i = 0 ; i < 8 ; i++) {
 /* Faz algo importante */
}
```

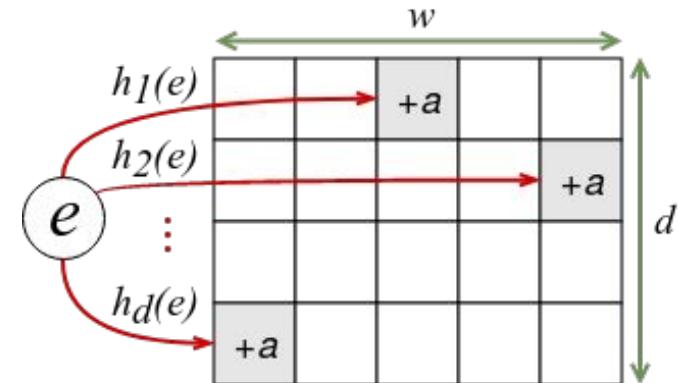
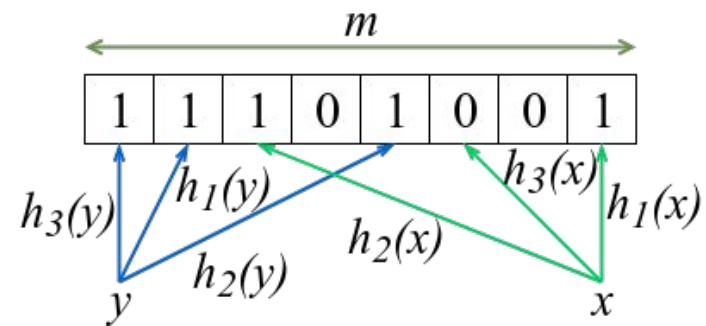
- Nesse caso, o código dentro do *for* é repetido 8 vezes no código objeto gerado

# Código fonte do eBPF no *kernel*

| Diretório                              | Descrição                                              |
|----------------------------------------|--------------------------------------------------------|
| <b>samples/bpf/</b>                    | Programas de exemplo (legado)                          |
| <b>tools/testing/selftests/bpf/</b>    | Programas de exemplo e testes unitários (mais recente) |
| <b>tools/lib/bpf/</b>                  | libbpf.h, AF_XDP e outras bibliotecas eBPF             |
| <b>kernel/bpf/</b>                     | Implementação de mapas, verificador, funções, etc.     |
| <b>include/uapi/linux/bpf.h</b>        | Biblioteca com as definições do eBPF                   |
| <b>net/core/filter.c</b>               | Núcleo da implementação                                |
| <b>arch/&lt;x86, arm, ...&gt;/net/</b> | Compilador JIT (algumas arquiteturas)                  |

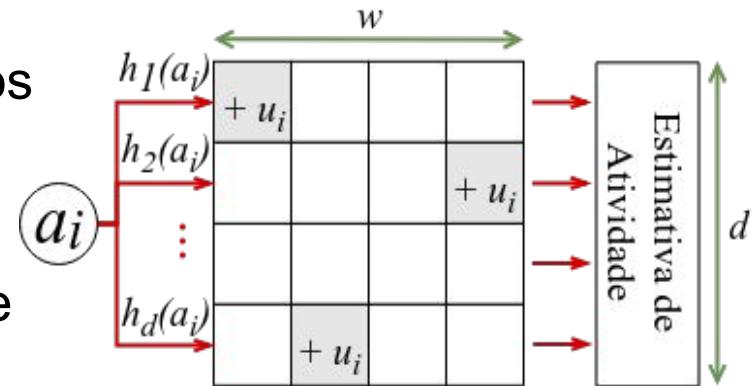
# Sketches

- Bloom Filter
  - Verifica se um elemento pertence a um conjunto
  - $m$  bits +  $k$  funções de hash
  - Operações de inclusão e verificação
- Count-Min Sketch
  - Contagem de frequência de eventos
  - Matriz  $w \times d$  +  $d$  funções de hash
  - Operações de atualização e estimativa



# Sketches

- k-ary Sketch
  - Contabiliza atividade de elementos
  - Matriz  $w \times d + d$  funções de hash
  - Operações de atualização, estimativa e combinação linear de sketches
- PCSA
  - Conta o número de elementos distintos de uma coleção grande
  - Matriz  $w \times d$
  - Operações de inserção e contagem



# Documentação

---

## Web

- [Código fonte do kernel do Linux](#)
- [Dive into BPF: a list of reading material](#)
- [Kernel Docs: networking/filter.txt](#)
- [BPF and XDP Reference Guide \(Cilium Project\)](#)
- [BPF Features by Linux Kernel Version \(BCC Project\)](#)
- [XDP Project \(Github\)](#)
- [Brendan Gregg's Blog \(eBPF para \*tracing e profiling\*\)](#)
- [Notes on BPF \(Parte 1/6\) - Oracle Linux Blog](#)

## Artigos

- [Creating Complex Network Services with eBPF: Experience and Lessons Learned](#)
- [The eXpress data path: fast programmable packet processing in the operating system kernel](#)