Análise de eficiência de subsistemas de filtragem e manipulação de pacotes no *kernel* do *Linux*

Gustavo Brito Sampaio

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2017

Gustavo Brito Sampaio

Análise de eficiência de subsistemas de filtragem e manipulação de pacotes no *kernel* do *Linux*

Monografia apresentada ao curso Ciência da Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Iguatemi E. Fonseca

Dezembro de 2017

Ficha Catalográfica elaborada por
Rogério Ferreira Marques CRB15/690

S192a Sampaio, Gustavo Brito.
Análise de eficiência de subsistemas de filtragem e manipulação de pacotes no *kernel* e *linux* / Gustavo Brito Sampaio. – João Pessoa, 2017.
45p. : il.

Monografia (Bacharelado em Ciência da Computação) – Universidade Federal da Paraíba - UFPB.
Orientador: Profº. Dr. Iguatemi Eduardo da Fonseca.

1. Sistema operacional. 2. Kernel. 3. Linux. 4. EBPF. I. Título.

UFPB/BSCI CDU: 004.451.9 (043.2)

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computaçãointitulado ***Análise de eficiência de subsistemas de filtragem e manipulação de pacotes no* kernel *do* Linux** de autoria de Gustavo Brito Sampaio, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Hamilton Soares da Silva
Universidade Federal da Paraiba

Prof. Dr. Fernando Menezes Matos
Universidade Federal da Paraiba

Prof. Dr. Iguatemi Eduardo Fonseca
Universidade Federal da Paraiba

Coordenador(a) do Departamento Informática
Gustavo Henrique Matos Bezerra Motta
CI/UFPB

João Pessoa, 19 de Dezembro de 2017

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

"A soul in tension that's learning to fly. Condition grounded but determined to try".

David Gilmore

DEDICATÓRIA

Dedico este trabalho aos meus pais Joaquim e Marcia e a minha irmã Karol, que me apoiaram ao longo deste caminho e que sempre acreditaram em mim independente de qualquer coisa.

AGRADECIMENTOS

Agradeço aos meus amigos Marcelo, Rafael Germano, Sane, Lidia, Juliana, Livia, Isabela, Jorismar, Arthur, Igor, Andy, Rafael Brayner, Vandre e Emerson por todos os momentos, alegrias e dificuldades, que passamos juntos dentro e fora da universidade. Agradeço também a Marcilio, Davysson e demais amigos e companheiros de laboratório.

RESUMO

O SeVen, ferramenta desenvolvida no Laboratório de Redes da UFPB, mostrou-se eficiente para filtragem na camada de aplicação e ataques do tipo *low-rate*, no entanto constatou-se que o mesmo não era verdade para ataques na categoria do tipo *flooding*. Em busca de aperfeiçoar a ferramenta, este trabalho visa analisar a eficiência de subsistemas no *kernel* do *linux* para filtragem e manipulação de pacotes e quadros de rede com o intuito de selecionar futuros subsistemas de base para a ampliação da capacidade da ferramenta. Para isto, foram analisados quatro subsistemas, *tc*, *xdp*, módulos do *kernel* usando o framework *netfilter* e o *iptables*. Tanto o *tc* quanto o *xdp*, funcionam usando uma tecnologia recentemente incorporada ao *kernel*, o *eBPF*, e utiliza uma máquina virtual para a execução segura e eficiente dos programas. Durante os testes foi observado uma superior eficiência no descarte de pacotes e quadros dos subsistemas baseados em *eBPF*, sendo este os escolhidos para a próxima versão do SeVen, enquanto que os subsistemas *netfilter* e *iptables* mostraram-se menos eficientes e com limitações de complexidade para o desenvolvimento.

Palavras-chave: kernel, linux, filtragem, eBPF, xdp

ABSTRACT

SeVen, a software developed in the Networking Laboratory of the Federal University of Paraiba, proved to be suitable for application-layer filtering and *low-rate* attacks, although it was found that the same was not true for flooding attacks. This work aims to analyze the efficiency of subsystems in the Linux kernel for filtering and manipulation of packets and network frames in order to select future base subsystems for the expansion of the software capacity. For this, four subsystems, tc, xdp, and kernel modules were analyzed using the netfilter framework and iptables. Both *tc* and *xdp* work by using a newly built kernel technology, eBPF, and uses a virtual machine for the safe and efficient execution of programs. During the tests it was observed a superior efficiency in the discarding of packages and frames of the subsystems based on eBPF, which were chosen for the next version of SeVen, while the netfilter and iptables subsystems were less efficient and with limitations of complexity for the development .

Key-words: kernel, linux, filtering, eBPF, xdp

LISTA DE FIGURAS

1 Linux: Arquitetura básica [1] 21

2 Fluxo de um pacote de rede no *netfilter* 21

3 Arquitetura simplificada do eBPF. Baseado em [3] 25

4 Arquitetura simplificada do controle de tráfego no *linux* 26

5 Arquitetura simplificada do *xdp* 28

6 Visão geral do ambiente de testes 29

7 Teste 1 - Recursos iptables 31

8 Teste 1 - Recursos nf (módulo) 32

9 Teste 1 - Recursos cls (eBPF) 32

10 Teste 1 - Recursos xdp (eBPF) 32

11 Teste 2 - Recursos iptables 34

12 Teste 2 - Recursos nf (módulo) 34

13 Teste 2 - Recursos cls (eBPF) 34

14 Teste 2 - Recursos xdp (eBPF) 35

15 Teste 3 - Recursos iptables 36

16 Teste 3 - Recursos nf (módulo) 36

17 Teste 3 - Recursos cls (eBPF) 36

18 Teste 3 - Recursos xdp (eBPF) 37

19 Teste 4 - Recursos iptables 38

20 Teste 4 - Recursos nf (módulo) 39

21 Teste 4 - Recursos cls (eBPF) 39

22 Teste 4 - Recursos xdp (eBPF) 39

23 Teste 5 - Recursos iptables 41

24 Teste 5 - Recursos nf (módulo) 41

25 Teste 5 - Recursos cls (eBPF) 41

26 Teste 5 - Recursos xdp (eBPF) 42

LISTA DE TABELAS

1 Códigos de retorno dos ganchos do *netfilter* 22

2 Códigos de retorno do gancho *xdp* 27

3 Teste 1 - Comparativo implementações 31

4 Teste 2 - Comparativo implementações 33

5 Teste 2 - Comparativo qualitativo 33

6 Teste 3 - Comparativo implementações 35

7 Teste 3 - Comparativo qualitativo 37

8 Teste 4 - Comparativo implementações 37

9 Teste 4 - Comparativo qualitativo 38

10 Teste 5 - Comparativo implementações 40

11 Teste 5 - Comparativo qualitativo 40

LISTA DE ABREVIATURAS

LaR – Laboratório de Redes

xdp – eXpress Data Path

tc – traffic control

NIC - Network Interface Card

skb - sk_buff

BPF - Berkeley Packet Filter

eBPF - extended BPF

cBPF - classic BPF

VM - Virtual Machine (Máquina virtual)

Conteúdo

1 INTRODUÇÃO **17**

1.1 Definição do Problema 17

1.2 Premissas e Hipóteses 18

1.2.1 Objetivo geral 18

1.2.2 Objetivos específicos 18

1.3 Estrutura da monografia 18

2 ARQUITETURA DOS SUBSISTEMAS **20**

2.1 Módulos do Kernel 20

2.2 iptables 22

2.2.1 Filtragem - Filter 23

2.2.2 Resolução de endereço - NAT 23

2.2.3 Mangle 24

2.2.4 Raw 24

2.3 Subsistema eBPF 24

2.4 Linux *Traffic Control* - TC 25

2.5 eXpress Data Path - xdp 26

3 METODOLOGIA **28**

4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS **31**

4.1 Teste 1 - DropAll 31

4.2 Teste 2 - DropTcpSubnet 33

4.3 Teste 3 - DropUDPSubnet 35

4.4 Teste 4 - DropTCPSubnetFlags 37

4.5 Teste 5 - DropTCPSubnetPayload 40

5 CONCLUSÕES E TRABALHOS FUTUROS **42**

REFERÊNCIAS **43**

1 INTRODUÇÃO

Estima-se que até 2020 irão existir conectados na rede mundial de computadores mais de 25 bilhões de dispositivos [19]. O crescimento exponencial da quantidade de dispositivos está acompanhado do aumento da quantidade de tráfego. Em junho de 2017, houve um pico histórico de 3 Tbps [17] no sistema IX.br, que promove, nas áreas metropolitanas, pontos de interconexão comercial e acadêmicos [6]. Além do volume considerado legítimo e orgânico, o crescimento da rede e da estrutura traz outro agravante à ser combatido. Ataques à rede cada vez mais elaborados estão surgindo a todo vapor. Ataques estes que impedem o funcionamento da rede no geral ou de um alvo específico. Em 2016, um ataque à servidores de DNS utilizando dispositivos da internet das coisas (IoT) deixou vários serviços indisponíveis. O ataque envolveu cerca de 100 mil de dispositivos conectados e gerou um tráfego de cerca de 1.2 Tbps [20]. Para atender ao crescimento e o grande volume de tráfego, cada vez mais faz-se necessário o aprimoramento e o aperfeiçoamento dos softwares e dos hardwares utilizados. Assim, é notável o esforço para se construir ambientes mais eficientes em todas as esferas (sistemas operacionais, cabeamento, etc).

1.1 Definição do Problema

Em parceria com a Rede Nacional de Ensino e Pesquisa (RNP), o Laboratório de Redes da Universidade Federal da Paraíba desenvolveu uma ferramenta capaz de mitigar ataques de negação de serviço na camada de aplicação do tipo *Low-Rate* [7] [13]. Durante o desenvolvimento foi detectado que a ferramenta, apesar de ter sua eficiência provada para ataques Low-Rate, não apresentava bons resultados para ataques de inundação - Ataques de inundação têm como objetivo esgotar recursos de rede atingindo o design da rede propriamente dita e não necessariamente fragilidades dos protocolos que percorrem [21].

Assim, em busca de formas de aperfeiçoar a ferramenta desenvolvida, este trabalho busca analisar a eficiência de medidas tomadas para aperfeiçoar a filtragem e tratamento de pacotes e quadros de rede em sistemas operacionais Linux. Mais especificamente foram analisados quatro subsistemas diretamente ligados ao *kernel* (núcleo do sistema operacional). Dois deles usando uma tecnologia recentemente incorporada ao *kernel* chamada *eBPF*, que é um *bytecode* executado por uma máquina virtual dentro do próprio *kernel*. Aqui testamos o *xdp* e *tc*. *xdp* é o *bytecode* aplicado em pontos chave dentro do próprio driver da placa de rede, ou seja antes de ser processado de fato pelo sistema operacional. Já o *tc* é o *bytecode* aplicado em pontos chave dentro do sistema de controle de tráfego do *kernel* - mais detalhes nas seções seguintes. O terceiro subsistema utiliza módulos do *kernel* e o último subsistema é o *iptables*, comumente conhecido como o *firewall* do Linux. Detectamos que muitas vezes o gargalo durante os ataques acontece nas camadas mais

iniciais do modelo OSI (Open System Interconnection), isso nos direcionou a investigar soluções também mais próximas.

1.2 Premissas e Hipóteses

Como citado, serão estudado quatro subsistemas capazes de filtrar e analisar quadros na camada 2 (*data link*). A primeira hipótese levantada é que o primeiro subsistema citado, o *xdp* tenha uma performance melhor quando comparado aos outros, por dois motivos, além de se tratar de um ambiente otimizado (*bytecode* em uma máquina virtual dentro do próprio *kernel*), está localizado dentro da placa de rede (NIC - Network Interface Card). Isto é, há o mínimo possível de alocação de recursos do sistema operacional para o processamento dos quadros. Apenas os recursos do *driver*. Além de ter o processamento na velocidade limite do dispositivo. No entanto, compatibilidade pode ser um problema, visto que é uma tecnologia recente, com a primeira aparição na versão 4.8 (2016) do *kernel*. A segunda hipótese a ser testada aqui é que o subsistema *tc*, também usando a tecnologia *eBPF* tenha uma eficiência melhor que o *iptables*. Apesar de usarem arquiteturas semelhantes, o *tc*, assim como o *xdp*, destaca-se por utilizar o ambiente otimizado da máquina virtual do *kernel* (mais detalhes nas seções seguintes). Uma terceira hipótese elaborada é a de que os módulos nativos do *kernel* tenham uma eficiência igual ou próximo ao subsistema *tc* ou no mínimo superior ao *iptables*, devido à simplificação da complexidade e do propósito.

1.2.1 Objetivo geral

Ajudar nas decisões de implementação e aperfeiçoamento de ferramentas para mitigar ataques de negação de serviço nas diversas camadas e aplicações.

1.2.2 Objetivos específicos

Selecionar, através de testes de performance, o melhor subsistema para servir de base para aperfeiçoamento da ferramenta SeVen. Testes de implementação e execução de quatro possíveis subsistemas no *kernel* do *linux* de forma a simular casos de filtragem reais.

1.3 Estrutura da monografia

- O Capítulo 2 descreve os principais conceitos e subsistemas utilizados em uma análise mais aprofundada;

- O Capítulo 3 descreve o processo de desenvolvimento das regras nos quatro subsistemas e como foram realizados os testes;
- O Capítulo 4 contém os resultados dos testes realizados nos subsistemas e nas regras implementadas;
- O Capítulo 5 trata das conclusões que podem ser extraídas dos testes realizados e discute trabalhos futuros a serem realizados.

2 ARQUITETURA DOS SUBSISTEMAS

Neste capítulo será descrito em detalhes o funcionamento e a arquitetura dos subsistemas usados neste trabalho. Assim, pode-se entender melhor as diferenças e semelhanças de cada subsistema e como podem ser utilizados para o propósito da evolução da ferramenta desenvolvida no LaR.

2.1 Módulos do Kernel

Módulos são peças essenciais no ecossistema do Linux. Eles são capazes de adicionar funcionalidades ao sistema sob demanda e sem a necessidade de recompilar todo o sistema ou até mesmo interromper a execução. Além disso, módulos trazem encapsulamento e abstração. Facilitando, assim, o desenvolvimento e manutenção. Um *driver* de acesso à disco, por exemplo, pode ser escrito em forma de módulo e só carregado ao sistema se o hardware estiver presente e a necessidade do *driver* for validada. Nos sistemas operacionais Linux, a arquitetura pode ser dividade em duas perspectivas. O espaço do usuário e o espaço do *kernel*. Essa divisão é importante para garantir segurança e estabilidade ao sistema como um todo, de forma que impede que processos em execução no espaço do usuário acessem regiões críticas ao funcionamento do sistema ou áreas sigilosas (senhas, memória de outros processos etc). Toda a comunicação entre o *kernel* e o usuário é feita através de chamadas do sistema. Na Figura 1 podemos observar melhor como é organizado. Módulos são escritos na linguagem C e são compilados em objetos que podem então ser carregados ao *kernel* em tempo de execução. Todos os módulos são executados no espaço do *kernel*. É importante destacar que esse fato traz uma série de complicações e complexidade para o desenvolvimento de um módulo. Uma falha em um módulo pode comprometer todo o sistema, não só afetando o funcionamento momentâneo, mas podendo comprometer permanentemente o sistema. Visto que não há qualquer tipo de validação, como a provida aos processos no espaço de usuário.

Em janeiro de 2001, com o lançamento do *kernel* 2.4, foi introduzido o framework chamado *netfilter*. Que trouxe diversas novas funcionalidades e melhorias ao sistema [5]. O framework consiste de de um conjunto de ganchos (*hooks*) que são chamados em momentos específicos do fluxo de dados no sistema operacional. Com o *netfilter* é possível realizar filtragem de pacotes, tradução de endereço (NAT) e porta (NAPT) etc.

No total, o *framework* contêm cinco *hooks*. São eles:

- NF_IP_PRE_ROUTING
- NF_IP_LOCAL_IN
- NF_IP_FORWARD

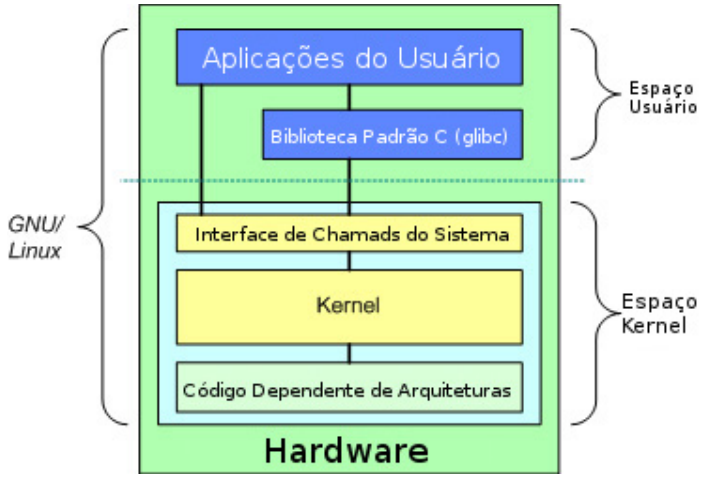


Figura 1: Linux: Arquitetura básica [1]

- NF_IP_LOCAL_OUT
- NF_IP_POST_ROUTING

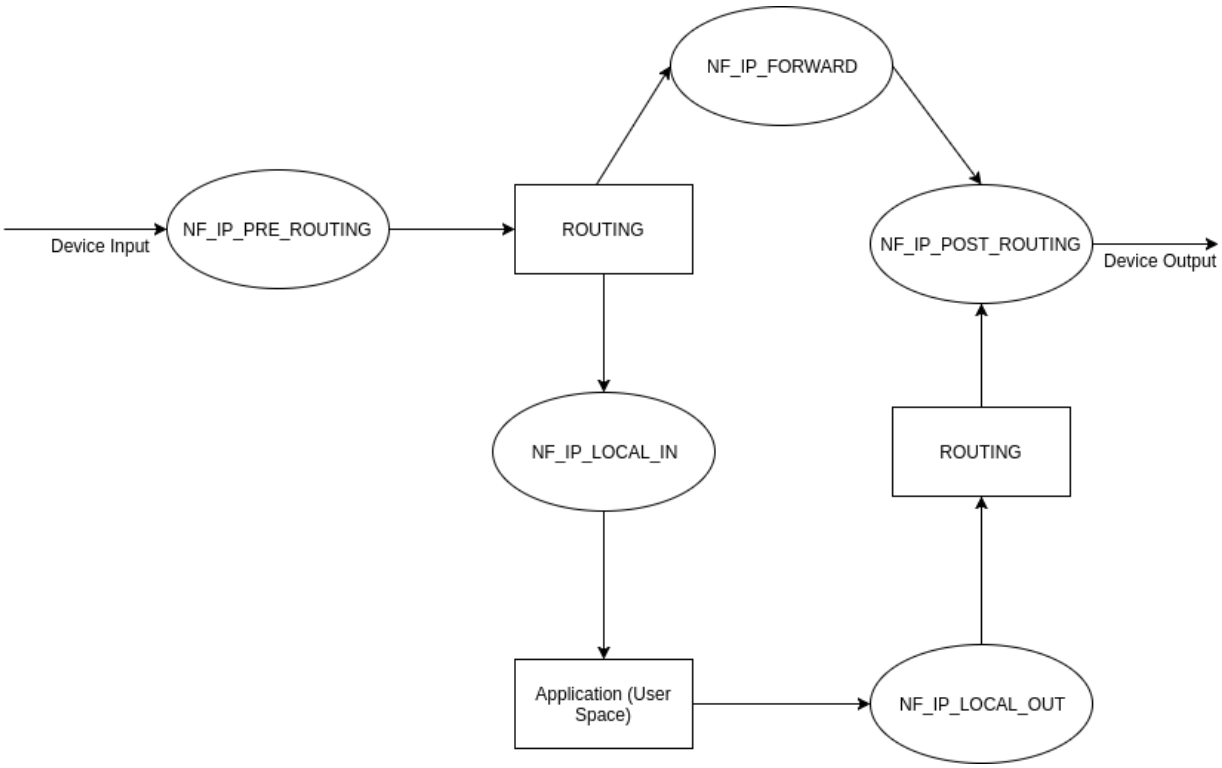


Figura 2: Fluxo de um pacote de rede no *netfilter*

A Figura 2 ilustra o fluxo percorrido pelos pacotes no *netfilter*. Após ser processado pela interface de rede (NIC), o pacote, ao entrar no *kernel*, aciona o primeiro *hook*: NF_IP_PRE_ROUTING. Neste *hook*, o sistema não realizou ainda decisões sobre rota, isto é se vai reencaminhar o pacote para outro destino ou aceitar e processar o mesmo. Caso decida redirecionar o pacote, outro *hook* é então ativado, o NF_IP_FORWARD. Logo em

seguida o NF_IP_POST_ROUTING. Caso o sistema decida aceitar o pacote, isto é o pacote tem como destino a máquina em questão, um outro gancho é ativado, o NF_IP_LOCAL_IN. Este gancho á acionado logo antes do pacote ser enviado para a aplicação destino no espaço do usuário. A aplicação irá então processar os dados recebidos. Quando uma aplicação envia um pacote o gancho análogo NF_IP_LOCAL_OUT é acionado e logo em seguida o gancho NF_IP_POST_ROUTING é também acionado antes de encaminhar para a NIC.

Os ganchos tem autonomia para decidir o que fazer com o pacote recebido para processamento. O retorno das funções gancho é usado para dizer ao *netfilter* qual a ação escolhida pelo módulo. A Tabela 1 descreve os retornos dos ganchos usados pelo *netfilter* e seus significados.

Tabela 1: Códigos de retorno dos ganchos do *netfilter*

Código	Significado
NF_DROP	O pacote deve ser descartado e os recursos devem ser liberados
NF_ACCEPT	O pacote é aceito e deve ser processado pelas próximas etapas do ciclo
NF_STOLEN	Informa ao *netfilter* que o módulo irá processar o pacote a partir de agora. I.E. O pacote não será mais encaminhado para as próximas etapas
NF_QUEUE	Envia o pacote para processamento no espaço do usuário
NF_REPEAT	O pacote é marcado para ser reavaliado pelo gancho

Os ganchos recebem como parâmetro, além das informações sobre os dispositivos de rede sendo usado, uma estrutura de dados chamada *sk_buff*. Esta estrutura é usada em todas as operações de rede dentro do *kernel* e contém o controle das informações recebidas e os dados propriamente dito. O *skb*, como é comumente referenciado, é organizado dentro de uma lista duplamente encadeada e pode ser manipulado diretamente para acesso aos dados ou com funções de apoio existentes nas bibliotecas do *linux*. Ex.: *ip_header* e *tcp_header*. O driver do dispositivo de rede é responsável por alocar na memória do *kernel* espaço suficiente para os novos *skbs* e inicializar valores quando necessário.

Na seção 3 será discutido a implementação do módulo usando o *netfilter* e suas particularidades.

2.2 iptables

Junto ao lançamento do *netfilter*, foi lançado o *iptables*. Um conjunto de aplicações para filtragem de pacotes em sistemas operacionais *linux*. As aplicações, que residem no

espaço do usuário, servem de interface para um módulo chamado *xtables*. O módulo utiliza os ganchos descritos na seção 2.1 para realizar a filtragem e o processamento. O conjunto *iptables* é conhecido como o *firewall* do *linux* e é famoso pela sua simplicidade e versatilidade. A suíte conta ainda com um sistema de rastreamento de controle de estados de pacotes chamado *conntrack*. Aumentando assim as possibilidades de filtragem dos pacotes. Todo o gerenciamento do *iptables* e do *xtables* é realizado através de regras. As regras podem monitorar, determinar uma ação, como nas descritas na Tabela 1 ou redirecionar pacotes. As regras são organizadas em cadeias e tabelas. As cadeias básicas por padrão são análogas aos ganchos definidos pelo *netfilter*. São elas:

- PREROUTING: NF_IP_PRE_ROUTING
- INPUT: NF_IP_LOCAL_IN
- FORWARD: NF_IP_FORWARD
- OUTPUT: NF_IP_LOCAL_OUT
- POSTROUTING: NF_IP_POST_ROUTING

Além destas, o *iptables* também permite a criação de cadeias personalizadas. O usuário pode organizar as regras em tabelas separadas e agrupa-las da maneira que achar mais interessante. As tabelas são definidas de acordo com a função das regras. Por padrão existem cinco tabelas implementadas. São elas: FILTER, NAT, MANGLE e RAW.

2.2.1 Filtragem - Filter

Nesta Tabela, as regras são capazes de decidir se um pacote deve ser mantido ou rejeitado no processamento. A maior parte das decisões de um *firewall* acontecem nesta Tabela. A filtragem pode levar em consideração os vários atributos de diferentes camadas de rede envolvidas no pacote. Por exemplo, uma regra pode ser construída de forma que um pacote com origem no endereço *ip* 192.168.5.4 e o marcador *SYN* do protocolo *tcp* ativado seja rejeitado. Nesta Tabela apenas as cadeias INPUT, FORWARD E OUTPUT são implementadas. Ou seja, todos os pacotes nesta tabelas já estão com a rota resolvida.

2.2.2 Resolução de endereço - NAT

A Tabela NAT contém regras para resolução de endereços nos pacotes. Assim elas irão determinar como e quando os pacotes serão manipulados. Normalmente a Tabela é usada em sistemas com redes privadas que não possuem acesso externo. Assim, faz-se necessário a tradução de endereços da rede externa para a rede interna. Nesta Tabela as cadeias PREROUTING, INPUT, OUTPUT e POSTROUTING estão presentes.

2.2.3 Mangle

A Tabela *mangle* é usada para realizar alterações nos cabeçalhos IP dos pacotes. Alguns dos atributos que podem ser alterados: MTU, TTL etc. Nesta Tabela todas as cadeias nativas estão presentes.

2.2.4 Raw

A Tabela *raw* é responsável por filtrar e marcar pacotes para que possam ser ignorados pelo sistema de rastreio *conntrack*. Reduzindo assim o consumo de recursos. Nesta Tabela apenas as cadeias PREROUTING e OUTPUT estão presentes.

Alguns sistemas ainda possuem uma Tabela específica para avaliar aspectos de segurança nos pacotes que entram e saem do sistema.

Na seção 3 é discutido a implementação e o uso de regras equivalentes aos outros subsistemas.

2.3 Subsistema eBPF

Em 1992 surgia uma tecnologia no *kernel* de sistemas *unix* chamada de Berkeley Packet Filter (BPF). A tecnologia foi concebida com o intuito de realizar a filtragem de pacotes de rede de forma mais eficiente. A tecnologia permite que aplicações no espaço do usuário injetem programas em ganchos no *kernel*. Os programas podem agir como filtros e podem rejeitar ou aceitar os pacotes recebidos. Os programas são executados em uma máquina virtual (VM) dentro do próprio *kernel*. Sendo assim, eles são verificados e, diferente dos módulos descritos na seção 2.1, existe uma garantia de que o código à ser executado não irá afetar negativamente o sistema operacional e nem afetar a segurança dos dados. Em dezembro de 2014, a versão 3.18 do *kernel* do *linux* introduzia o *extended BPF* (*eBPF*). Uma versão estendida do *BPF*. Essa atualização deu a capacidade de injetar programas, compilados em *bytecode* da máquina virtual, em diversos eventos do sistema, não só eventos relacionados a rede e tanto no espaço do usuário quanto no espaço do *kernel*. Como por exemplo chamadas do sistema, funções etc. Essa atualização ampliou os métodos de monitoramento e análise de performance em sistemas *linux*.

Os programas *eBPF* são acoplados e desacoplados aos eventos através de um programa no espaço do usuário. Na versão do 3.18 do *kernel* também foi introduzido estruturas de dados especiais que são usadas para compartilhar dados entre os programas *eBPF* e os programas no espaço do usuário. Essas estruturas, aqui chamadas de mapas, podem ser criadas de forma que sejam sincronizadas entre CPUs ou distribuídas entre CPUs, sendo necessárias as devidas operações para manipular os dados. A leitura e escrita das

mapas pode ser feita de forma assíncrona pelo programa no espaço do usuário, isto é o programa no espaço do usuário pode realizar a operações nos mapas sem interromper a execução do *bytecode* dentro do *kernel*. A Figura 3 ilustra de forma simplificada o processo de construção e acoplagem. O código fonte é compilado em *bytecode* específico da VM, logo em seguida é verificado se não há nenhuma irregularidade e por fim é executado pela VM nos ganchos definidos pelo processo no espaço do usuário. Os programas ainda se comunicam com o processo no espaço do usuário através dos mapas.

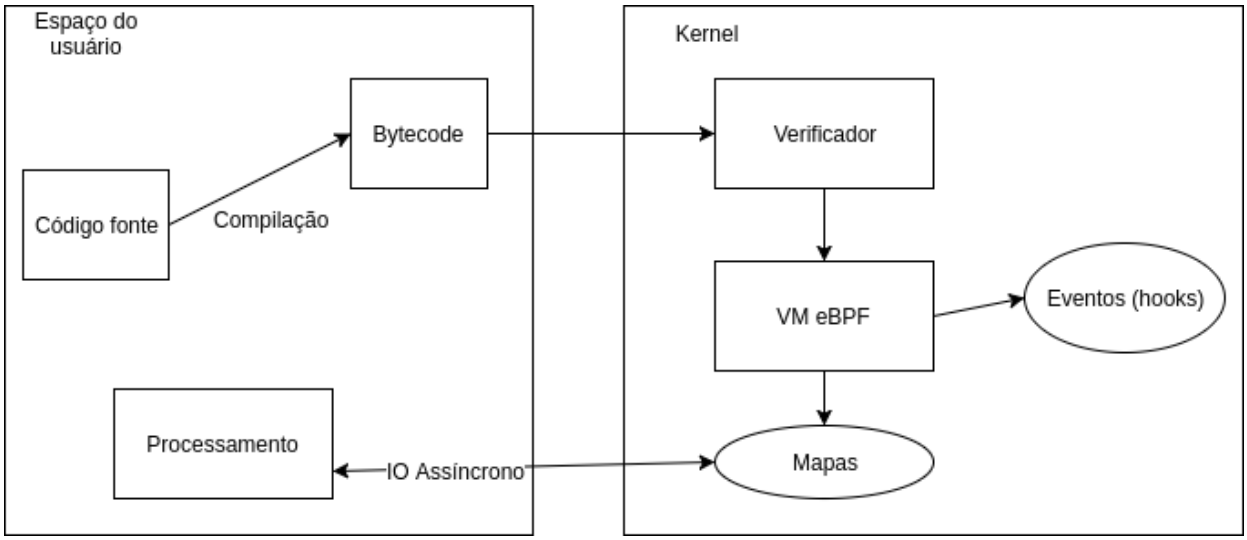


Figura 3: Arquitetura simplificada do eBPF. Baseado em [3]

Os dois últimos subsistemas utilizados aqui neste trabalho, *Linux Traffic Control* (seção 2.4) e *xdp* (seção 2.5), utilizam esta tecnologia como base e serão discutidas nas próximas seções.

2.4 Linux *Traffic Control* - TC

Entre as versões 2.2 e 2.4 foi desenvolvido e incorporado ao *kernel* o subsistema de controle de tráfego (*traffic control* - tc) e junto com o *netfilter* (seção 2.1), veio para alavancar o processamento e filtragem de pacotes de rede no sistemas operacionais *linux*. O subsistema consiste de estruturas de enfileiramento. Os pacotes, tanto na saída para o NIC quanto na entrada pelo NIC, são enfileirados e processados pelo *tc*. O processamento pode determinar se um pacote vai ser aceito ou recusado, pode determinar a velocidade de enfileiramento e a retirada de pacotes e há ainda uma série de técnicas de manipulação de qualidade de serviço (QoS), como por exemplo filas de prioridade. Isso garante uma maior compatibilidade e estabilidade na comunicação com o NIC e com o *link* de rede. Um caso de uso bem comum é atribuir maior prioridade, reservando banda de transmissão, para aplicações especificas. No *tc* é possível configurar, para cara interface de rede, qual tipo de enfileiramento deve ser utilizado, aqui chamados de *qdisc* (Queuing Disciplines). Os

pontos de ação do controle de tráfego são chamados de *ingress* e *egress*. Operam na entrada e na saída de pacotes respectivamente. *qdiscs* aplicados na entrada funcionam como filtros, i.e. aplicam regras de policiamento. Já os *qdiscs* aplicados na saída funcionam como enfileiramento, controle de banda, priorização etc. A Figura 4 ilustra os pontos de ação.

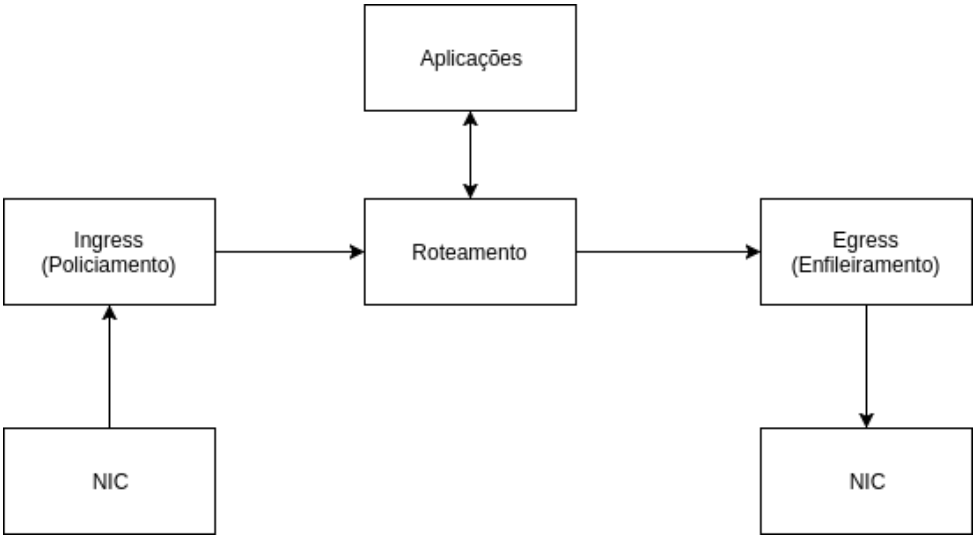


Figura 4: Arquitetura simplificada do controle de tráfego no *linux*

No *tc* existem ainda outras estruturas que complementam o seu funcionamento. Classes e Filtros são usados em conjuntos. Os filtros são aplicados nos pacotes e os classificam usando as classes. Através das classes é possível definir grupos de prioridade, por exemplo. Filtros também são responsáveis pelo policiamento dos pacotes. Além de atribuir classes aos pacotes, os filtros podem atribuir ações.

Na maioria dos sistemas *linux*, o *qdisc* de saída padrão é o *pfifo_fast*. Este *qdisc* possui três filas de prioridades. Os pacotes vão sendo classificados e enfileirados para transmissão e apenas após a primeira fila ser esvaziada, é que as demais são processadas [15]. Neste trabalho é usado um outro tipo de *qdisc* chamado *clsact*. Este tipo de estrutura permite o uso de programas criados usando *ebpf* para policiarmos os pacotes no ponto de entrada. Com a opção *direct_action* ativada neste *qdisc*, o programa escrito é capaz de retornar qual ação deve ser tomada para os pacotes recebidos [12] e assim diminuir gargalos.

2.5 eXpress Data Path - xdp

O último subsistema a ser apresentado neste trabalho é o *Express Data Path* (*xdp*). O processamento de pacotes/quadros neste subsistema é feito no ponto mais baixo na pilha de processamento. Ganchos são aplicados diretamente no driver do dispositivo de rede. Assim, toda a filtragem é feita antes de qualquer alocação de estruturas no *kernel*,

como por exemplo os *skbs* (ver seção 2.1). Assim toda a pilha de processamento de rede do *kernel* fica sujeita ao retorno do processamento pelos ganchos *xdp*. Isso inclui os subsistemas citados nas seções anteriores. Sem a alocação e o processamento do *kernel*, espera-se que este subsistema tenha uma performance melhor. Para o funcionamento, não é necessário hardware ou dispositivo especial, no entanto o driver para o dispositivo precisa ter implementado os ganchos *xdp*. A arquitetura básica para o *xdp* está presente no *kernel* desde a versão 4.8, no entanto, até a finalização deste trabalho, apenas 11 drivers possuem suporte aos ganchos, dentre eles podemos destacar o *virtio_net* e o Intel *e1000* driver. O primeiro presente na plataforma de virtualização KVM e o segundo um dispositivo virtual bastante usado no sistema VMware. Ambos são drivers nativos do *linux*. Alguns drivers para dispositivos Mellanox também estão disponíveis.

Assim como o *netfilter* e o *tc*, o *xdp* consiste de um programa compilado em bytecode (eBPF) para a máquina virtual do *kernel*. Semelhante aos outros subsistemas, os programas *xdp* podem modificar os pacotes recebidos. Aqui, diferente dos subsistemas anteriores, apenas pacotes de entrada são processados. O retorno do programa é usado para determinar o destino do pacote recebido. A Tabela 2 lista os retornos e os destinos associados. A Figura 5 ilustra o posicionamento do gancho *xdp*.

Tabela 2: Códigos de retorno do gancho *xdp*

Código	Ação
XDP_PASS	O pacote está liberado para ser processado pela pilha de rede do *kernel*
XDP_DROP	O pacote é descartado pelo driver
XDP_TX	Informa ao driver reprocessar aquele pacote
XDP_ABORTED	Utilizado em caso de erro no programa *xdp*. Resulta em descarte

Apesar de ser uma tecnologia nova, tecnologias usando *eBPF* tem-se mostrado bastante promissor. O maquinário, no entanto, ainda precisa de cuidados, bem como a integração com os atuais sistemas de mitigação como citado por Gilberto Bertin [11] ao introduzir *xdp* nos ambientes de proteção da Cloudflare e Jakub Kicinski e Nicolaas Viljoen [16] ao comparar o *traffic control* e o *xdp*.

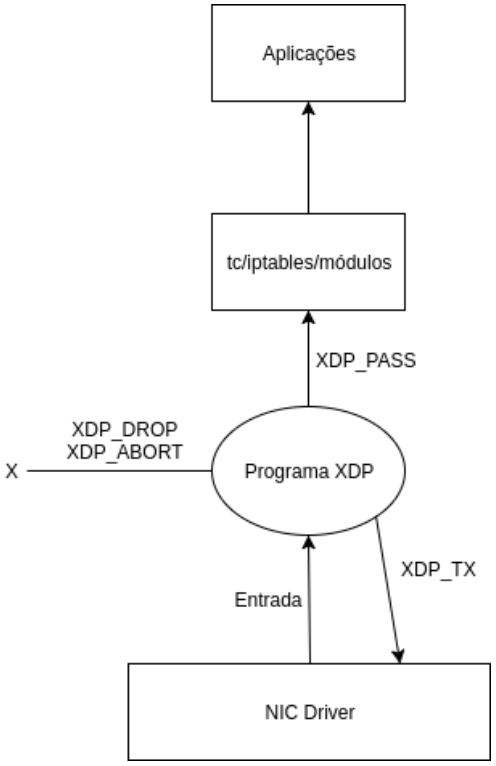


Figura 5: Arquitetura simplificada do *xdp*

3 METODOLOGIA

Para a avaliação dos subsistemas foram implementadas uma série de programas e regras com objetivos semelhantes nos quatro subsistemas. Para os testes foram utilizados duas ferramentas de geração de pacotes. A primeira é um módulo do *kernel*, chamado *pktgen*, para geração de pacotes em alta velocidade. Aqui o módulo é controlado por um *script*, que pode ser encontrado nos arquivos de fonte do *kernel* [2]. O *script* inicia *threads* a nível do *kernel* para aumentar a eficiência da geração. O *pktgen* é comumente usado para testar a eficiência de NICs durante o desenvolvimento e é capaz de sobrecarregar uma NIC comum de baixa capacidade. Embora bastante eficiente, o *pktgen* torna bastante complexo a manipulação dos pacotes. A segunda ferramenta foi o já conhecido *hping* [9]. Com esta última ferramenta podemos manipular de forma prática e eficiente os pacotes gerados. Emulando protocolos, alterando endereço, porta, *payload* etc.

O ambiente construído para os testes consiste de uma máquina virtual usando a distribuição Linux Ubuntu 14.04 64bits com um *kernel* na versão 4.7 customizado com o o patch contendo o *driver* do dispositivo *e1000* atualizado com suporte à eBPF e xdp. A máquina virtual é criada sobre o *hypervisor KVM* e gerenciada pela ferramenta *vagrant*. A máquina virtual contém duas interfaces de rede. Uma gerenciada pelo *vagrant*, chamada de *eth0* com uma rede virtual gerenciada automaticamente e usada para controle da máquina e acesso via *ssh*. E a segunda, chamada de *eth1*, com endereço estático usada

para os testes. Além disso a VM possui 2.0G de memória ram e um processador baseado na série Intel Xeon E3-12xx v2 (Ivy Bridge). A Figura 6 ilustra o ambiente e a configuração das interfaces de rede.

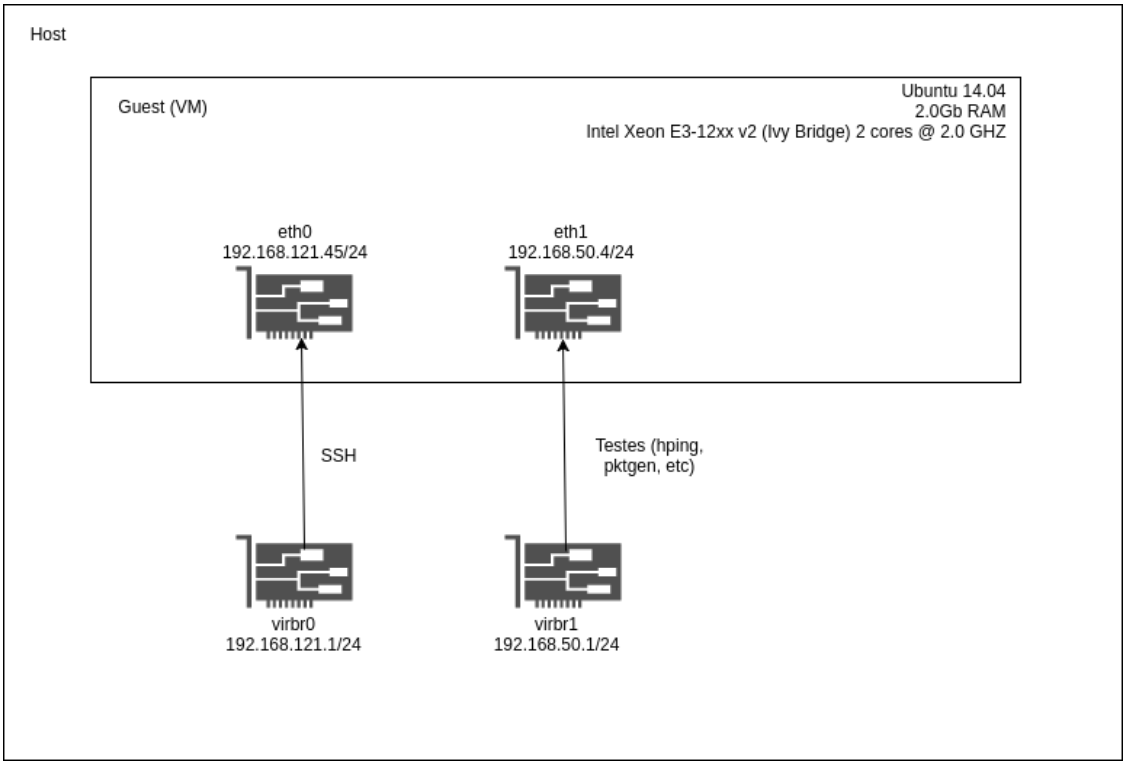


Figura 6: Visão geral do ambiente de testes

Todo o ambiente foi baseado no ambiente de testes da comunidade IO Visor [8], algumas modificações foram necessárias para que os outros subsistemas funcionassem de forma correta. O código de configuração do ambiente com as devidas customizações pode ser encontrado online no Github do autor [4].

As estatísticas foram coletadas usando um *script* desenvolvido especialmente para este trabalho. O *script* além de memória e cpu, também coleta as seguintes estatísticas tanto na entrada quanto na saída da interface de testes (*eth1*):

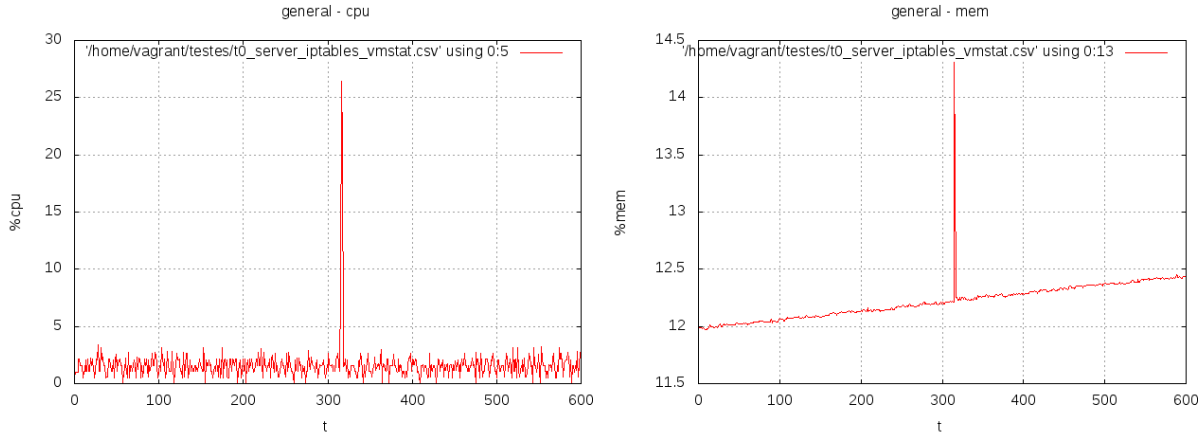
- Pacotes por segundo
- Kilobytes por segundo
- Kilobytes totais
- Pacotes totais

Além disso, quando possível, foi realizado a coleta de estatísticas direto no subsistema. No caso das aplicações usando *ebpf*, foi utilizado mapas por cpu para o armazenamento de estatísticas e posteriormente, já no espaço do usuário, os mapas de cada

cpu foram agrupados. Já no caso do módulo do *kernel*, uma *thread* a nível de *kernel* foi implementada para a coleta das estatísticas.

Além do *script* de coleta de estatísticas, foi desenvolvido também um *script* para controle de execução das ferramentas de teste. Este *script* foi necessário para iniciar mais de um processo ao mesmo tempo e limitar o tempo de execução dos mesmos.

Por fim, para complementar e trazer uma visão melhor da qualidade de serviço, foi utilizado a ferramenta *ping*, para medir o tempo de resposta da máquina durante os testes.



(a) CPU **(b) Memória**

Figura 7: Teste 1 - Recursos iptables

4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS

No total foram realizados cinco testes de 10 minutos. Cada teste com um conjunto de regras e procedimentos diferentes. Os cinco testes foram realizados nos quatro subsistemas de forma semelhante.

4.1 Teste 1 - DropAll

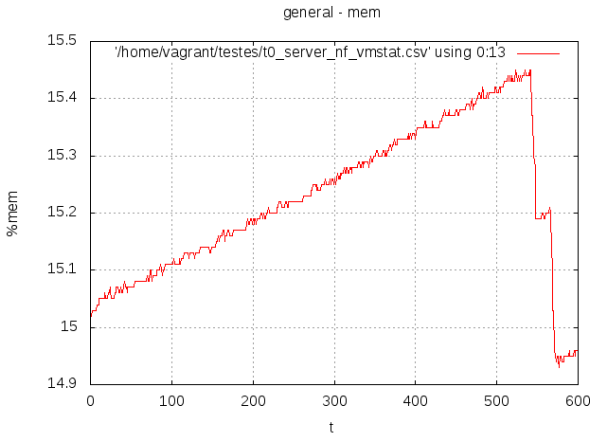
Neste teste foi utilizado a ferramenta *pktgen* para gerar pacotes. Foi utilizada uma instância enviando pacotes udp para o endereço ip da interface de testes (192.168.50.4). Cada pacote gerado contém 18 bytes de *payload*. Neste teste, os subsistemas não levam em consideração nenhuma característica do pacote, ou seja o máximo de pacotes possíveis deve ser descartado. A Tabela 3 ilustra os resultados obtidos nos três subsistemas onde foi implementado código. Podemos notar que o *xdp* obteve uma quantidade de pacotes descartados bem maior, bem como o pico de pacotes por segundo.

Tabela 3: Teste 1 - Comparativo implementações

Subsistema	Pacotes totais	Pico Pkts/s
nf (módulo)	43391643	87228
cls	43950287	90741
xdp	48580162	98316

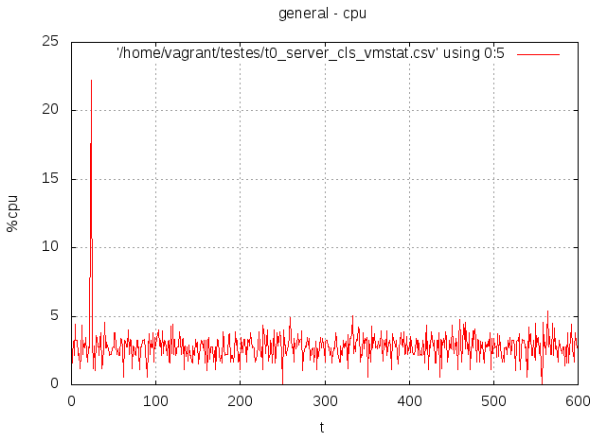
A regra para descartar todos os pacotes no subsistema *iptables*, conseguiu um total de 39220375 descartes. Valor bem inferior aos demais subsistemas. Quanto ao uso de memória e cpu, os quatro subsistemas tiveram resultados próximos.

(a) CPU

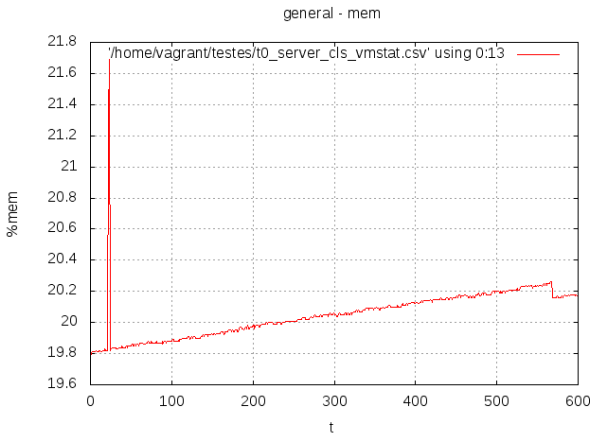


(b) Memória

Figura 8: Teste 1 - Recursos nf (módulo)

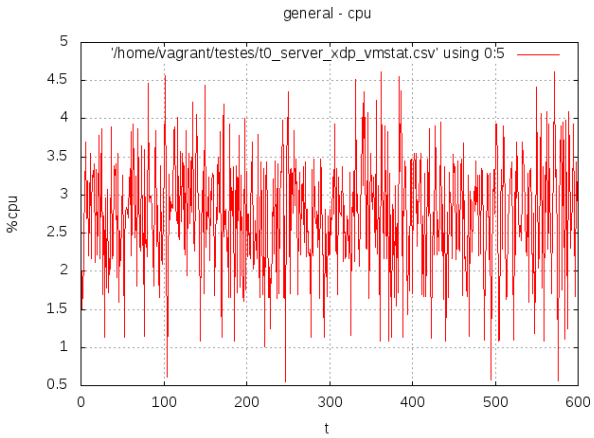


(a) CPU

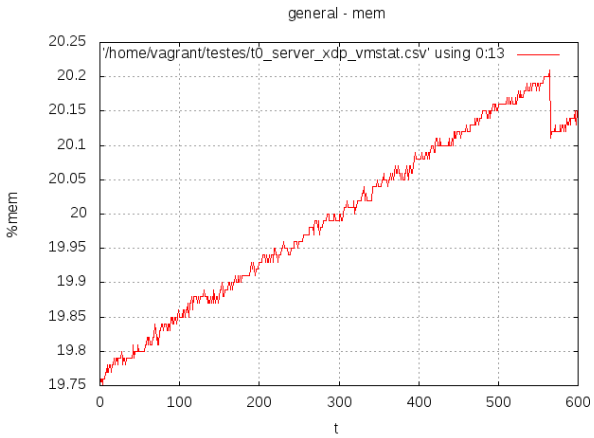


(b) Memória

Figura 9: Teste 1 - Recursos cls (eBPF)



(a) CPU



(b) Memória

Figura 10: Teste 1 - Recursos xdp (eBPF)

4.2 Teste 2 - DropTcpSubnet

Neste teste foi utilizado a ferramenta *hping* para gerar pacotes. Foram utilizadas cinco instâncias enviando pacotes tcp para a porta 80 para o endereço ip da interface de testes (192.168.50.4). Os pacotes foram forjados com endereços de origem aleatórios (spoofing). Cada pacote continha apenas a *flag* tcp *SYN* ativado e um *payload* de 64 bytes. Neste teste, os subsistemas devem filtrar pacotes com qualquer origem, mas com o destino sendo um endereço pertencente a subrede em que a interface estava contida, neste caso 192.168.50.0/24. A Tabela 4 mostra os resultados obtidos nos três subsistemas onde foi implementado código. Podemos notar que tanto o *cls* quanto o *xdp* possuem uma eficiência maior, visto que conseguiram descartar mais pacotes que o módulo. O *xdp* ainda apresentou um pico de pacotes por segundo maior que o *cls*.

Tabela 4: Teste 2 - Comparativo implementações

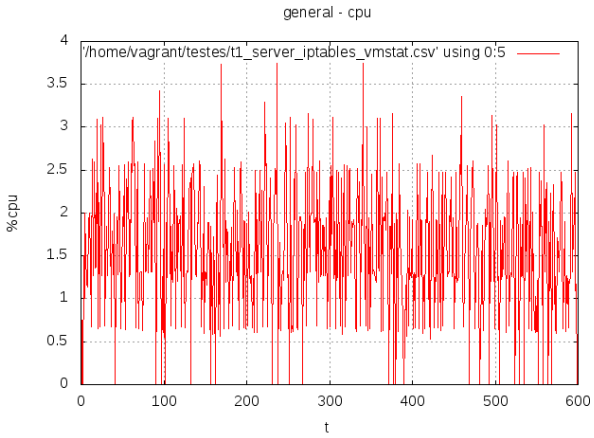
Subsistema	Pacotes totais	Pico Pkts/s
nf (módulo)	10081802	21737
cls	10303476	24175
xdp	10293136	26364

A Tabela 5 agrupa as coletas obtidas pela ferramenta *ping* durante os testes. Nela podemos observar o tempo mínimo, médio e máximo de resposta capturados durante os testes, bem como a porcentagem de pacotes perdidos. Podemos observar que os subsistemas tiveram resultados médios parecidos.

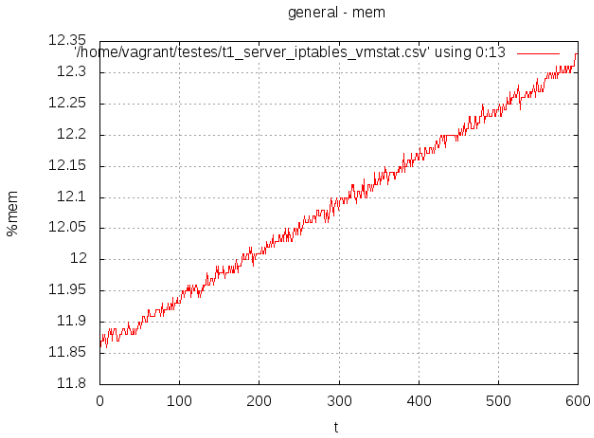
Tabela 5: Teste 2 - Comparativo qualitativo

Subsistema	rtt min	rtt médio	rtt max	% perdidos
nf (módulo)	0.551	51.008	158.483	0
cls	0.603	51.720	192.889	0
xdp	0.913	52.553	124.193	0
iptables	0.913	52.512	219.076	0

Quanto ao uso de memória e cpu, os quatro subsistemas tiveram resultados próximos.

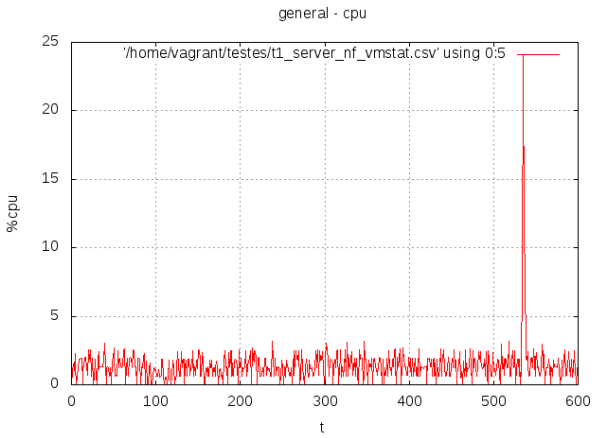


(a) CPU

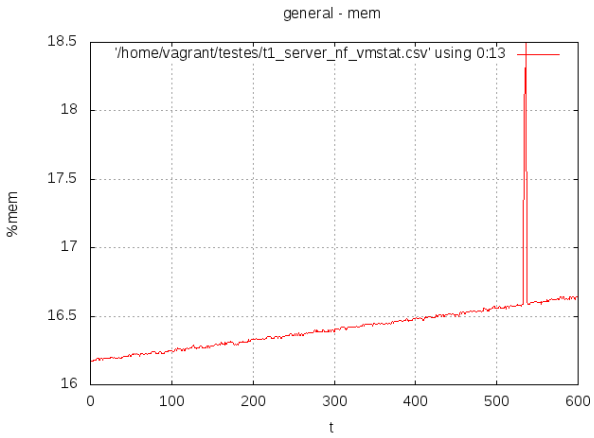


(b) Memória

Figura 11: Teste 2 - Recursos iptables

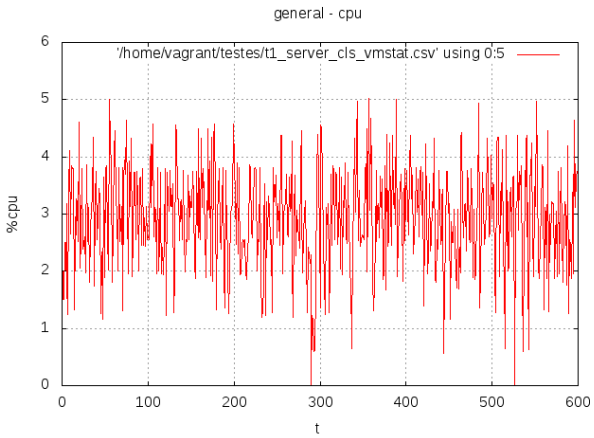


(a) CPU

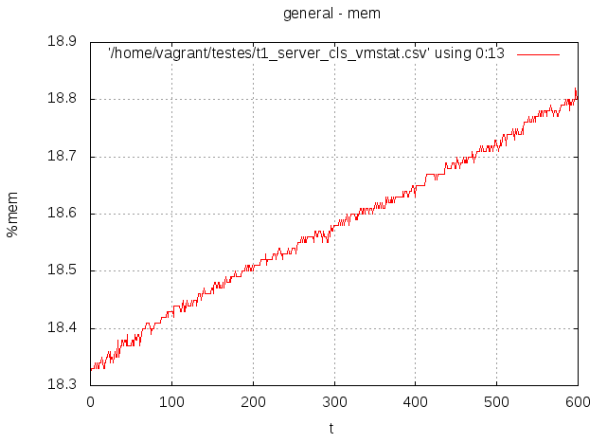


(b) Memória

Figura 12: Teste 2 - Recursos nf (módulo)

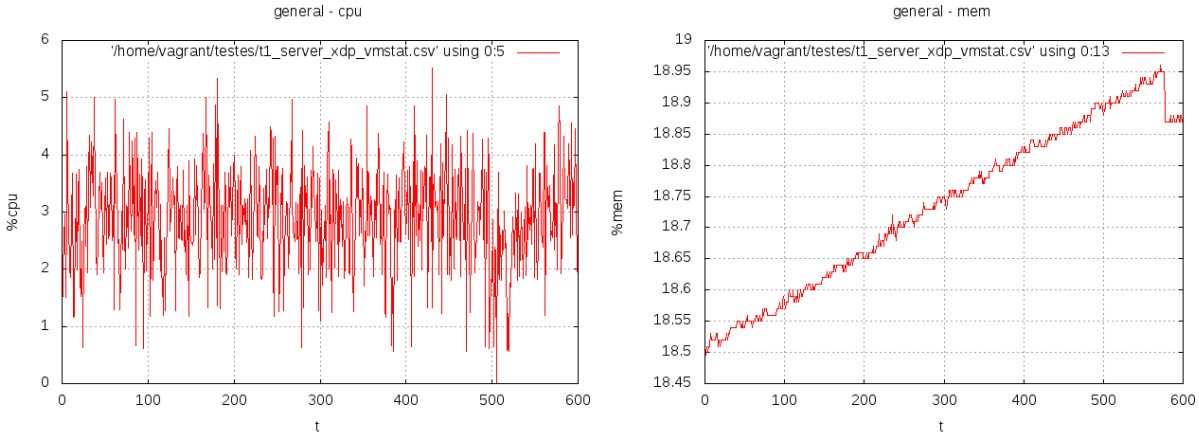


(a) CPU



(b) Memória

Figura 13: Teste 2 - Recursos cls (eBPF)



(a) CPU **(b) Memória**

Figura 14: Teste 2 - Recursos xdp (eBPF)

4.3 Teste 3 - DropUDPSubnet

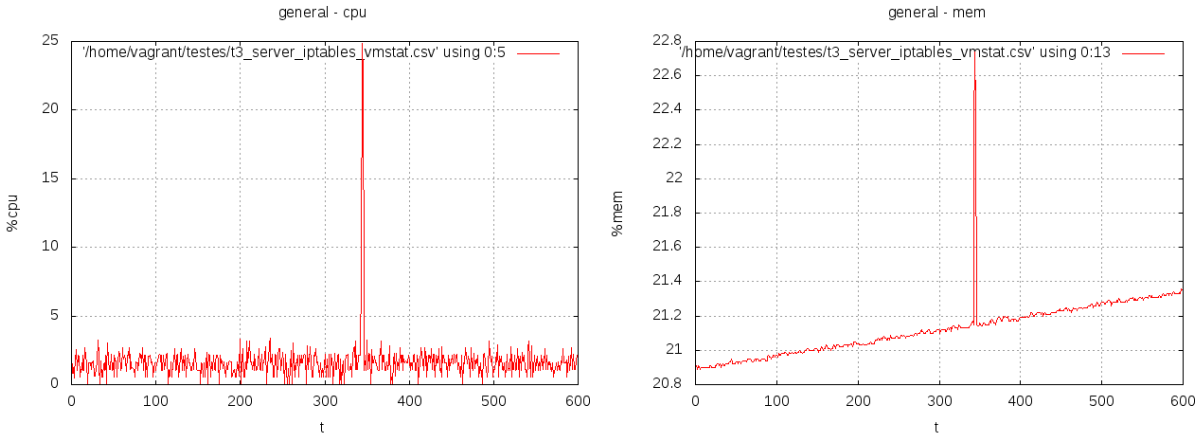
Neste teste foi utilizado a ferramenta *pktgen* para gerar pacotes. Foi utilizada uma instância enviando pacotes udp para o endereço ip da interface de testes (192.168.50.4). Cada pacote gerado contém 18 bytes de *payload*. Neste teste, os subsistemas devem filtrar pacotes udp com qualquer origem, mas com o destino sendo um endereço pertencente a subrede em que a interface estava contida, neste caso 192.168.50.0/24. A Tabela 6 ilustra os resultados obtidos nos três subsistemas onde foi implementado código. Podemos notar que o *xdp* obteve uma quantidade de pacotes descartados bem maior, bem como o pico de pacotes por segundo.

Tabela 6: Teste 3 - Comparativo implementações

Subsistema	Pacotes totais	Pico Pkts/s
nf (módulo)	37273236	83866
cls	38357076	86210
xdp	42880738	87847

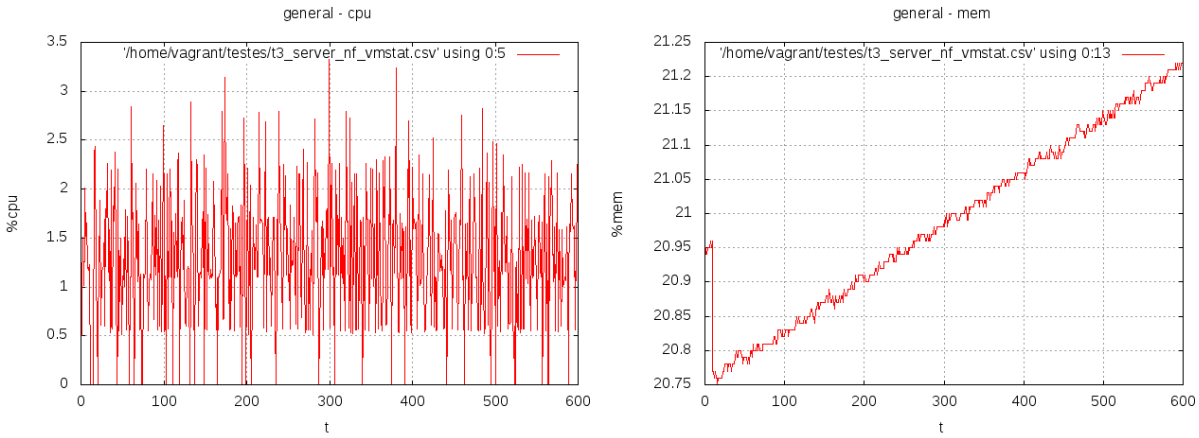
A Tabela 7 agrupa as coletas obtidas pela ferramenta *ping* durante os testes. Nela podemos observar o tempo mínimo, médio e máximo de resposta capturados durante os testes, bem como a porcentagem de pacotes perdidos.

Quanto ao uso de memória e cpu, os quatro subsistemas tiveram resultados próximos.



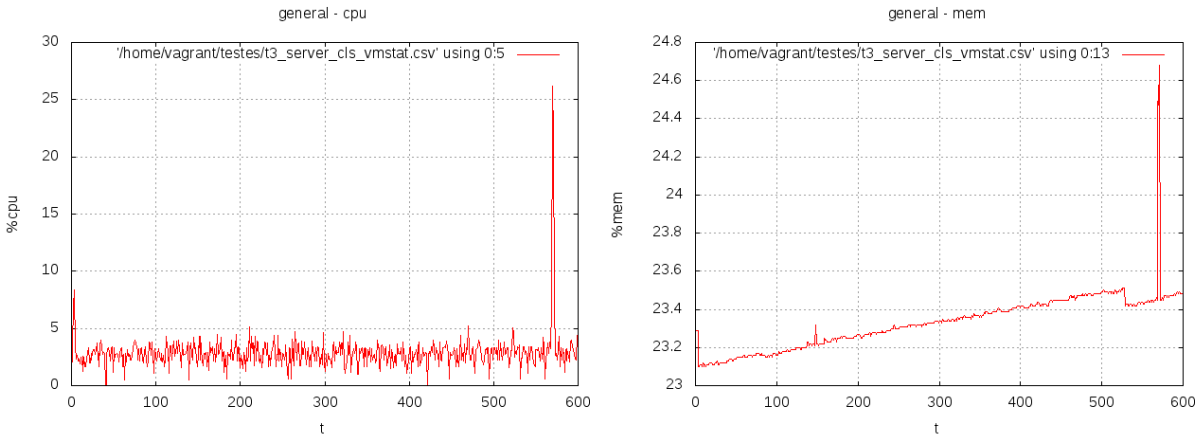
(a) CPU **(b) Memória**

Figura 15: Teste 3 - Recursos iptables



(a) CPU **(b) Memória**

Figura 16: Teste 3 - Recursos nf (módulo)

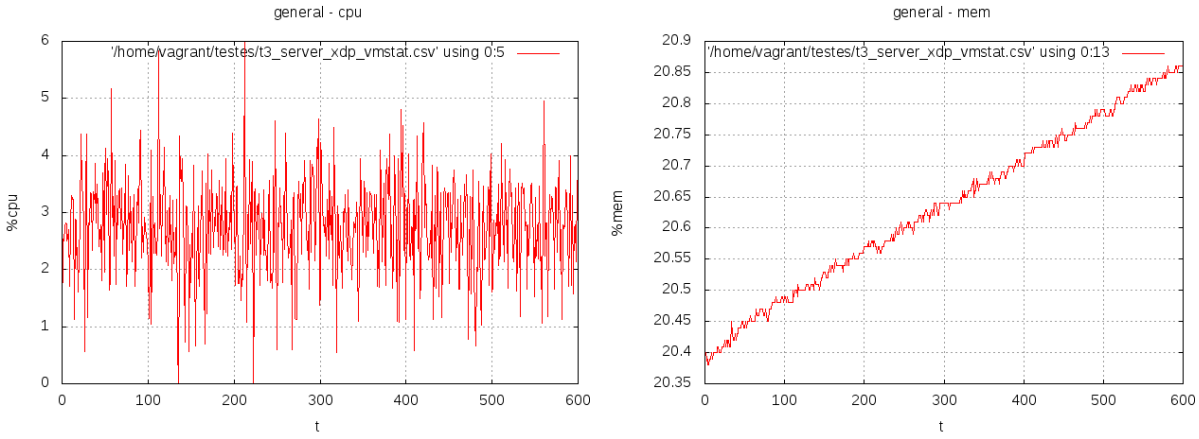


(a) CPU **(b) Memória**

Figura 17: Teste 3 - Recursos cls (eBPF)

Tabela 7: Teste 3 - Comparativo qualitativo

Subsistema	rtt min (ms)	rtt médio (ms)	rtt max (ms)	% perdidos
nf (módulo)	0.986	59.142	1110.461	92
cls	1.000	59.228	1054.618	92
xdp	0.878	97.866	2081.063	89
iptables	0.976	116.103	2106.694	92



(a) CPU **(b) Memória**

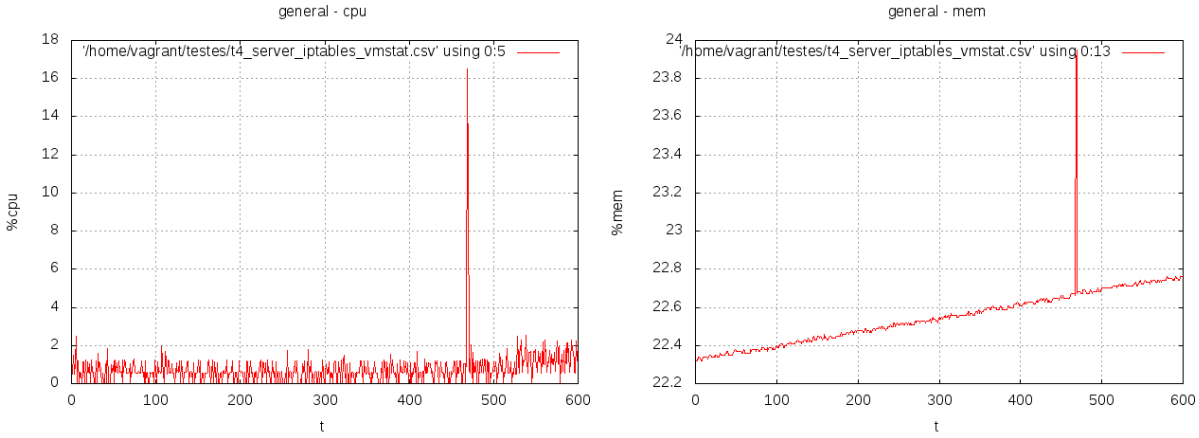
Figura 18: Teste 3 - Recursos xdp (eBPF)

4.4 Teste 4 - DropTCPSubnetFlags

Neste teste foi utilizado a ferramenta *hping* para gerar pacotes. Foram utilizadas oito instâncias enviando pacotes tcp para a porta 80 e para o endereço ip da interface de testes (192.168.50.4). Os pacotes foram forjados com endereços de origem aleatórios (spoofing). Cada pacote continha as flags tcp *SYN* e *RST* ativadas e um *payload* de 64 bytes. Neste teste, os subsistemas devem filtrar pacotes tcp com as flags *SYN* e *RST* ativadas, de qualquer origem, mas com o destino sendo um endereço pertencente a subrede em que a interface estava contida, neste caso 192.168.50.0/24. A Tabela 8 ilustra os resultados obtidos nos três subsistemas onde foi implementado código. Podemos notar que novamente o *xdp* apresentou uma eficiência maior, visto que conseguiu descartar mais pacotes e ainda apresentou um pico de pacotes por segundo maior.

Tabela 8: Teste 4 - Comparativo implementações

Subsistema	Pacotes totais	Pico Pkts/s
nf (módulo)	24782656	57163
cls	23799010	59785
xdp	31686091	75657



(a) CPU **(b) Memória**

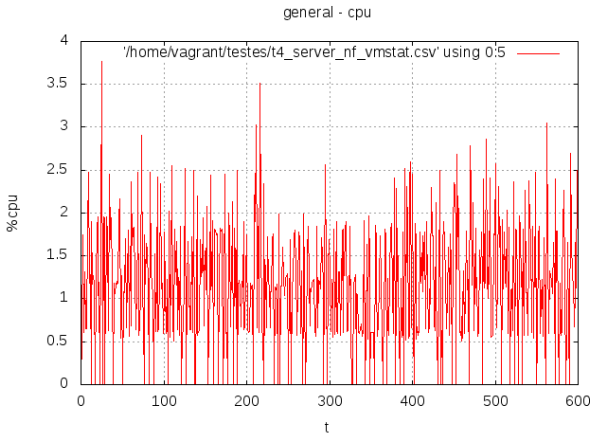
Figura 19: Teste 4 - Recursos iptables

A Tabela 9 agrupa as coletas obtidas pela ferramenta *ping* durante os testes. Nela podemos observar o tempo mínimo, médio e máximo de resposta capturados durante os testes, bem como a porcentagem de pacotes perdidos.

Tabela 9: Teste 4 - Comparativo qualitativo

Subsistema	rtt min (ms)	rtt médio (ms)	rtt max (ms)	% perdidos
nf (módulo)	0.460	30.520	1096.054	41
cls	0.572	41.087	2093.438	38
xdp	0.544	19.494	87.352	32
iptables	0.508	52.837	2093.914	54

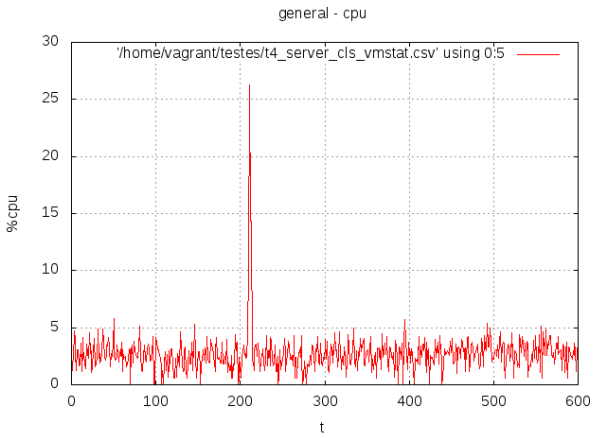
Quanto ao uso de memória e cpu, os quatro subsistemas tiveram resultados próximos.



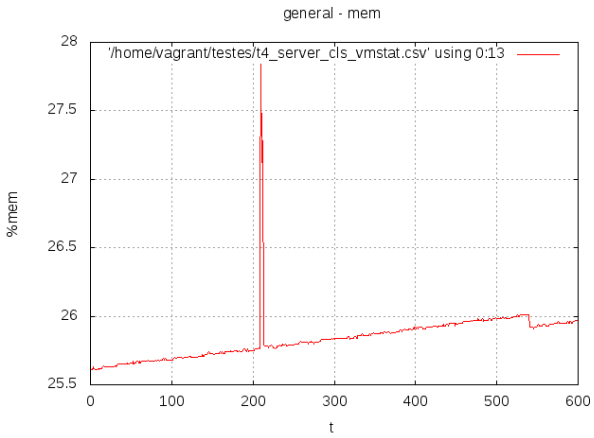
(a) CPU

(b) Memória

Figura 20: Teste 4 - Recursos nf (módulo)

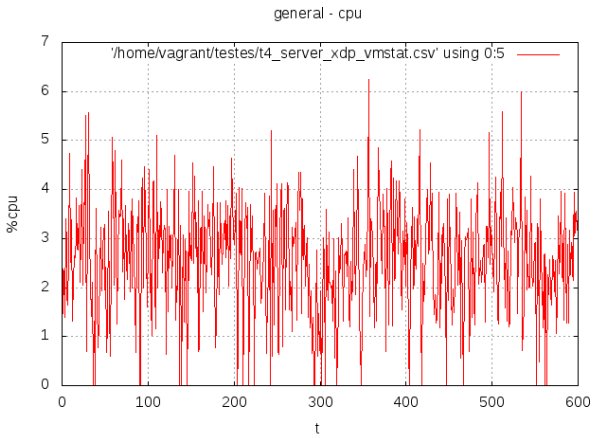


(a) CPU

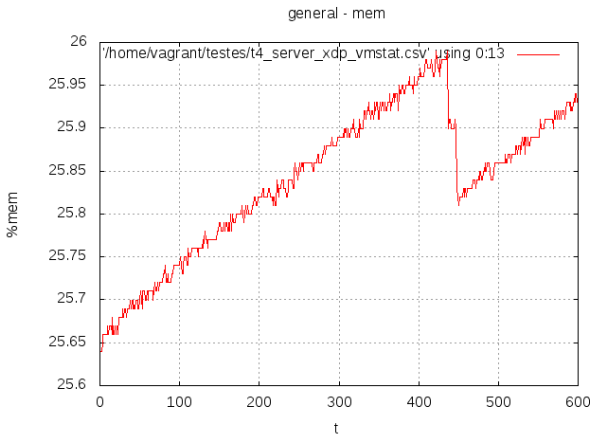


(b) Memória

Figura 21: Teste 4 - Recursos cls (eBPF)



(a) CPU



(b) Memória

Figura 22: Teste 4 - Recursos xdp (eBPF)

4.5 Teste 5 - DropTCPSubnetPayload

Neste teste foi utilizado a ferramenta *hping* para gerar pacotes. Foram utilizadas oito instâncias enviando pacotes tcp para a porta 80 e para o endereço ip da interface de testes (192.168.50.4). Os pacotes foram forjados com endereços de origem aleatórios (spoofing). Cada pacote continha as flags tcp *SYN* e *RST* ativadas e um *payload* de 64 bytes. Diferente dos demais, neste teste o payload contém um marcador que será usado para filtragem. Assim, neste teste os subsistemas devem filtrar pacotes tcp de qualquer origem, com o destino sendo um endereço pertencente a subrede em que a interface estava contida, neste caso 192.168.50.0/24 e com o marcador presente no *payload*. Nos testes foi usado o caractere X no segundo byte do *payload* como marcador. A Tabela 10 ilustra os resultados obtidos nos três subsistemas onde foi implementado código. Novamente, o *xdp* mostrou-se mais eficiente que os demais subsistemas, tanto em quantidade de pacotes, quanto no pico de pacotes por segundo.

Tabela 10: Teste 5 - Comparativo implementações

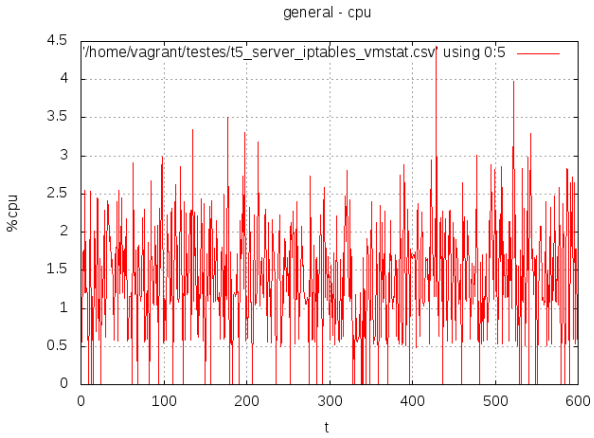
Subsistema	Pacotes totais	Pico Pkts/s
nf (módulo)	24281934	55978
cls	23301456	53643
xdp	29051250	70629

A Tabela 11 agrupa as coletas obtidas pela ferramenta *ping* durante os testes. Nela podemos observar o tempo mínimo, médio e máximo de resposta capturados durante os testes, bem como a porcentagem de pacotes perdidos.

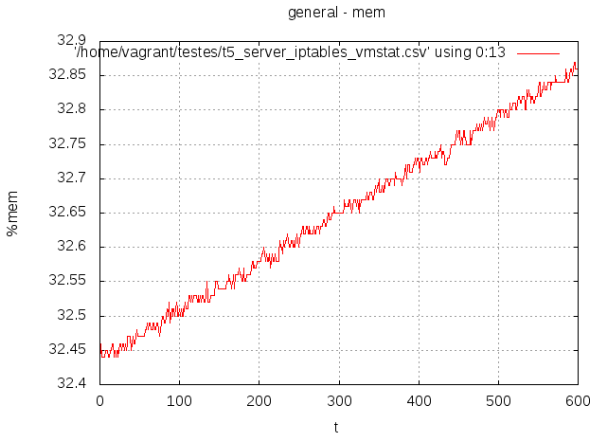
Tabela 11: Teste 5 - Comparativo qualitativo

Subsistema	rtt min (ms)	rtt médio (ms)	rtt max (ms)	% perdidos
nf (módulo)	0.297	20.533	79.082	16
cls	0.312	28.448	1336.599	26
xdp	0.256	11.717	63.800	8
iptables	0.248	32.081	1003.851	29

Quanto ao uso de memória e cpu, os quatro subsistemas tiveram resultados próximos.



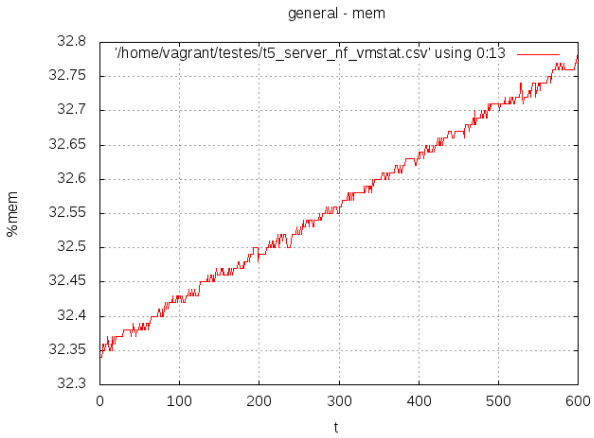
(a) CPU



(b) Memória

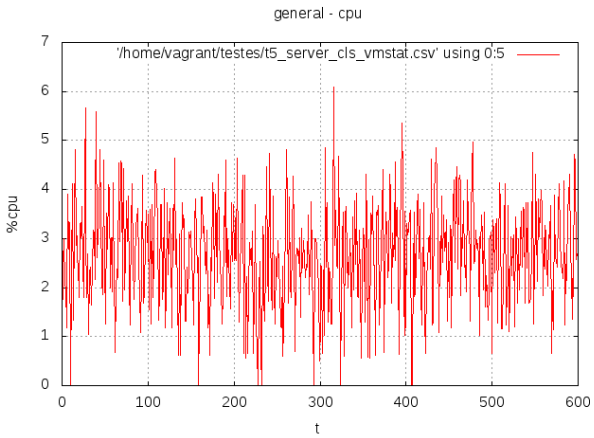
Figura 23: Teste 5 - Recursos iptables

(a) CPU

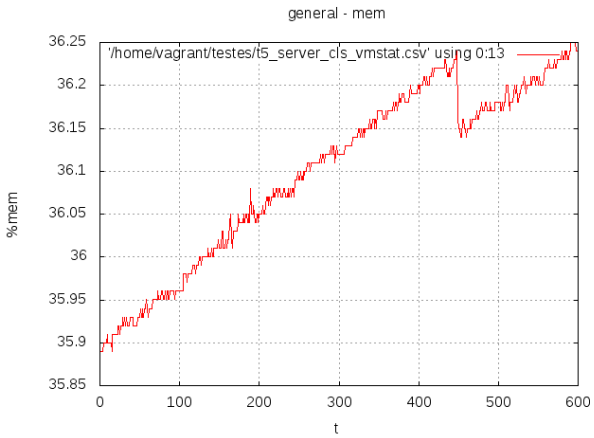


(b) Memória

Figura 24: Teste 5 - Recursos nf (módulo)

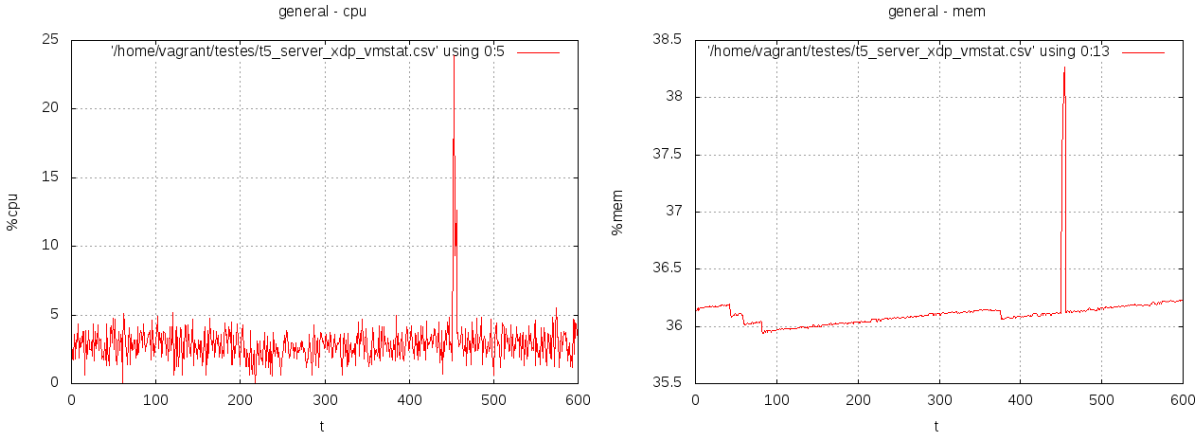


(a) CPU



(b) Memória

Figura 25: Teste 5 - Recursos cls (eBPF)



(a) CPU **(b) Memória**

Figura 26: Teste 5 - Recursos xdp (eBPF)

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como objetivo avaliar a eficiência em virtude da evolução da complexidade das regras escritas em subsistemas de filtragem e análise de pacotes à nível de *kernel* em sistemas operacionais *linux* e sugerir um subsistema para a implementação de uma futura versão do sistema SeVen desenvolvido no Laboratório de Redes da UFPB (LaR). As etapas para os testes foram relatadas neste trabalho com o objetivo de tornar-las reproduzíveis em futuros trabalhos e garantir a imparcialidade dos resultados. A metodologia adotada mostrou-se eficiente e conseguiu ampliar a visão e o entendimento dos subsistemas, tanto dos resultados finais quanto dos aspectos inerentes ao uso e ao maquinário dos mesmos.

Durante a construção dos programas em *eBPF*, foi-se considerado utilizar a biblioteca oficial do framework *bcc* na linguagem de programação Go para a manipulação e injeção dos objetos compilados na máquina virtual do *kernel*. No entanto, posteriormente foi-se observado que a mesma não possuía suporte para programas *xdp*. Aqui destaca-se como contribuição a implementação do suporte *xdp* na biblioteca em questão.

Além disso, pode-se notar que os programas usando o subsistema em *ebpf*, em especial os programas *xdp*, obtiveram uma eficiência melhor ou igual aos programas construídos usando módulos do *kernel*. Observou-se também que o subsistema *iptables*, mostrou-se inferior ou igual ao subsistema de módulos. Acredita-se que o maquinário presenta no *iptables* que o tornam capaz de executar operações de filtragem genéricas trazem um custo em cima das próprias operações. Como visto, o crescimento do volume de tráfego nas grandes redes requer que as aplicações e os procedimentos sobre os dados trafegados, sejam os mais eficientes possíveis. Notadamente, levando em consideração este aspecto, recomenda-se o uso de tecnologias usando *ebpf* para tais necessidades.

Como citado na seção 2.3, os programas *eBPF* durante a compilação para o código da máquina virtual do *kernel* e execução, passam por um processo de verificação. Com a verificação, os programas recebem uma garantia a mais de que não irão realizar operações indevidas. Isso trouxe uma pico na curva de aprendizado e na depuração de erros nos programas, uma vez que as mensagens de erro não apresentavam informações claras. No entanto, após a evolução do aprendizado, os obstáculos relacionados ao verificador tornam-se menos frequentes. Em comparação com os módulos, a falta de um verificador trouxe dificuldades ainda maiores no desenvolvimento dos testes. Erros afetaram o uso do sistema, e em certas ocasiões ocasionando falhas críticas. Tendo em vista que a diferença do consumo de recursos nos programas usando *eBPF* com o verificador e no módulo é mínima (como observados nos resultados da seção 4), recomenda-se novamente o uso do subsistema *eBPF*.

Outro ponto importante a ser levantado é quanto a versão mínima com suporte ao subsistema *eBPF*, tanto o *tc* quanto ao *xdp*. Como citado na seção 2.3, o suporte no *kernel* surgiu na versão 3.18. Já o *tc* com o *qdisc clsact* (seção 2.4, foi introduzido na versão 4.5 e o *xdp* apenas na versão 4.8 (como visto na seção 2.5). Considerando ainda o suporte necessário nos drivers dos dispositivos para o *xdp*. Essa fragmentação de versões do *kernel* e pequena adoção das novas versões do *kernel*, tornam a tecnologia imatura de certa forma. O que acaba dificultando a adoção. No entanto, a evolução do *kernel* e o crescimento da adoção das versões suportadas tornam válida novamente a utilização do subsistemas em *eBPF*. Com uma ressalva, portar a ferramenta SeVen tanto para o módulo quanto para *eBPF* irão garantir uma abrangência maior de sistemas suportados.

Como trabalhos futuros, planeja-se o desenvolvimento de uma primeira versão da ferramenta SeVen usando os subsistemas *xdp* e *tc*. Nessa etapa será avaliado o funcionamento do algorítimo usado atualmente frente a nova arquitetura requerida pelos subsistemas. Além disso é sugerido a análise de mais um conjunto de bibliotecas capaz de rápido processamento de pacotes. O DPDK [10] também é capaz disso, mas com o diferencial de poder ser executado no espaço do usuário.

REFERÊNCIAS

[1] Anatomy of the linux kernel. Disponível em: <https://www.ibm.com/developerworks/library/l-linux-kernel/index.html>, Acesso em: 27 out 2017.

[2] Codigo fonte do kernel e do script controlador do pktgen. Disponível em: <https://github.com/torvalds/linux/blob/master/samples/pktgen/pktgen_sample03_burst_sing Acesso em: 30 out 2017.

[3] ebpf tools - brendan gregg. Disponível em: <http://www.brendangregg.com/ebpf.html>, Acesso em: 30 out 2017.

[4] Github do ambiente de testes - lowrider. Disponível em: <https://github.com/GustavoKatel/lowrider>, Acesso em: 15 nov 2017.

[5] Netfilter website. Disponível em: <https://www.netfilter.org/>, Acesso em: 27 out 2017.

[6] Projeto ix.br. Disponível em: <https://ix.br/intro>, Acesso em: 26 out 2017.

[7] Projetos gt-ufpb. Disponível em: <http://lar-ufpb.net/?page_id=40>, Acesso em: 26 out 2017.

[8] Página da comunidade io visor. Disponível em: <https://www.iovisor.org/>, Acesso em: 13 nov 2017.

[9] Página do hping. Disponível em: <https://github.com/antirez/hping>, Acesso em: 30 out 2017.

[10] Site oficial dpdk. Disponível em: <http://dpdk.org/>, Acesso em: 19 nov 2017.

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

[12] Daniel Borkmann. On getting tc classifier fully programmable with cls bpf. 2016. Netdev1.1 Conference.

[13] Yuri Gil Dantas, Vivek Nigam, and Iguatemi E. Fonseca. An adaptive selective defense for application layer DDoS attacks. In *CTDSeg*, 2016. Selected as a Candidate for the Prize of Best Master Thesis in Security in Brazil.

[14] M. Hoogesteger, R. d. O. Schmidt, and A. Pras. Itsa: Internet traffic statistics archive. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 995–996, April 2016.

[15] Pasquale Imputato and Stefano Avallone. An analysis of the impact of network device buffers on packet schedulers through experiments and simulations. *Simulation Modelling Practice and Theory*, 80(Supplement C):1 – 18, 2018.

[16] Jakub Kicinski and Nicolaas Viljoen. ebpf hardware offload to smartnics: cls bpf and xdp.

[17] Cristina De Luca. Troca de tráfego diária da internet brasileira atinge pico 3 tbps. Disponível em: <https://porta23.blogosfera.uol.com.br/2017/06/20/troca-de-trafego-diaria-da-internet-brasileira-atinge-pico-3-tbps>, Acesso em: 26 out 2017.

[18] SOBRENOME Nome. Título. Disponível em: <http://example.com>, Acesso em: dia (não incluir o zero à esquerda) mês (usar abreviações) ano.

[19] Christy Pettey. The internet of things and the enterprise. Disponível em: <https://www.gartner.com/smarterwithgartner/the-internet-of-things-and-the-enterprise/>, Acesso em: 26 out 2017.

[20] Nicky Woolf. Ddos attack that disrupted internet was largest of its kind in history, experts say. Disponível em: <https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet>, Acesso em: 26 out 2017.

[21] Jian Yuan and K. Mills. Monitoring the macroscopic effect of ddos flooding attacks. *IEEE Transactions on Dependable and Secure Computing*, 2(4):324–335, Oct 2005.