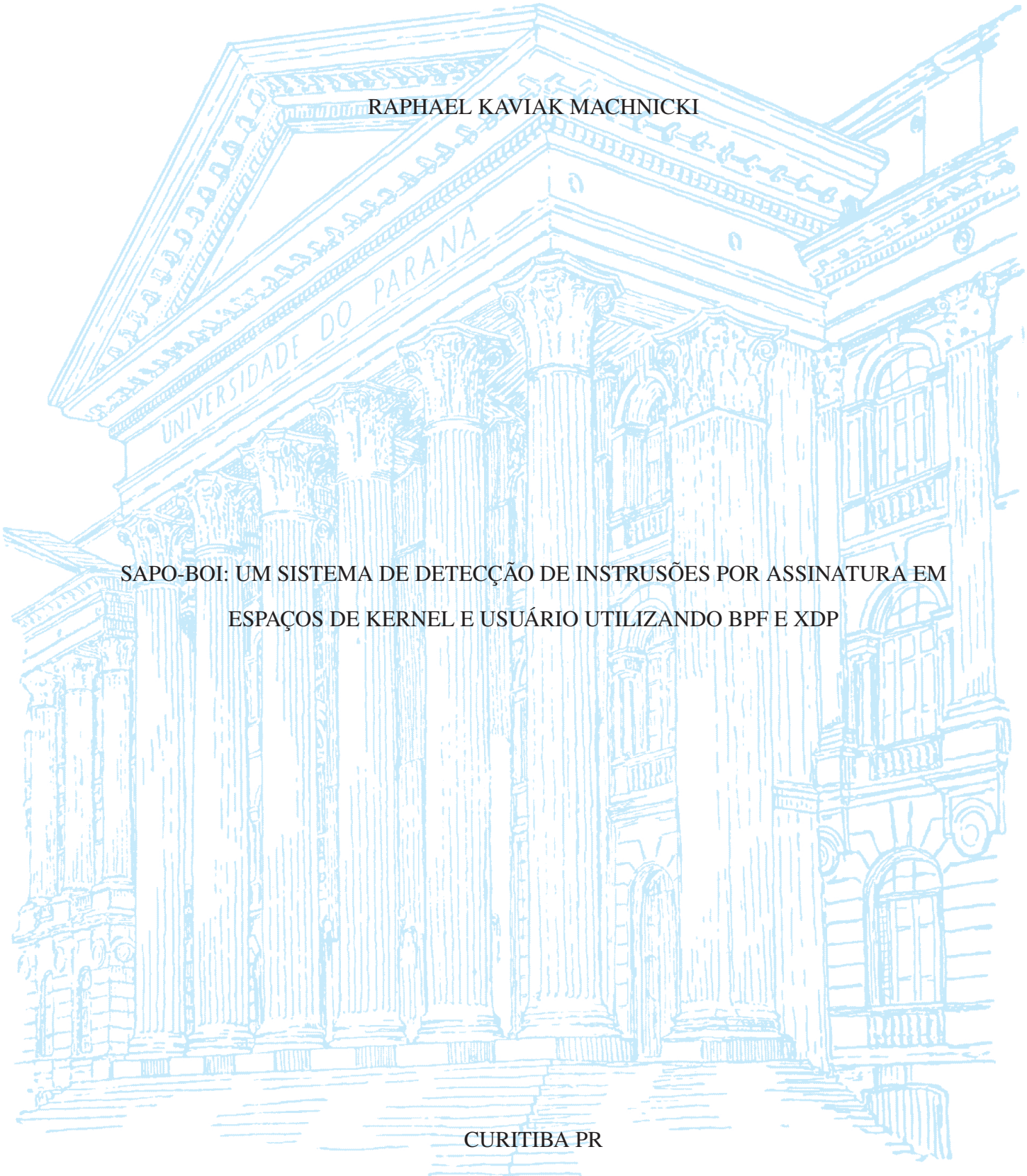
UNIVERSIDADE FEDERAL DO PARANÁ

RAPHAEL KAVIAK MACHNICKI



SAPO-BOI: UM SISTEMA DE DETECÇÃO DE INSTRUSÕES POR ASSINATURA EM ESPAÇOS DE KERNEL E USUÁRIO UTILIZANDO BPF E XDP

CURITIBA PR

2025

RAPHAEL KAVIAK MACHNICKI

SAPO-BOI: UM SISTEMA DE DETECÇÃO DE INSTRUSÕES POR ASSINATURA EM ESPAÇOS DE KERNEL E USUÁRIO UTILIZANDO BPF E XDP

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática, no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: André Ricardo Abed Grégio.

Coorientador: Vinicius Fulber-Garcia.

CURITIBA PR

2025

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA CIÊNCIA E TECNOLOGIA

Machnicki, Raphael Kaviak
Sapo-boi: um sistema de detecção de intrusões por assinatura em espaços de kernel e usuário utilizando BPF e XDP. / Raphael Kaviak Machnicki. – Curitiba, 2025.
1 recurso on-line : PDF.

Dissertação (Mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas. Programa de Pós-Graduação em Informática.

Orientador: André Ricardo Abed Grégio.
Coorientador: Vinicius Fulber-Garcia.

1. Sistemas de detecção de intrusão. 2. Programa BPF (Berkeley Packet Filter). 3. Soquetes XDP (eXpress Data Path). I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Grégio, Ricardo Abed. IV. Fulber-Garcia. Vinicius V. Título.

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **RAPHAEL KAVIAK MACHNICKI**, intitulada: **Sapo-boi: Um sistema de detecção de intrusões por assinatura em espaços de kernel e usuário utilizando BPF e XDP**, sob orientação do Prof. Dr. ANDRÉ RICARDO ABED GRÉGIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 18 de Junho de 2025.

Assinatura Eletrônica
30/06/2025 11:48:57.0
ANDRÉ RICARDO ABED GRÉGIO
Presidente da Banca Examinadora

Assinatura Eletrônica
27/06/2025 20:52:41.0
ELIAS PROCÓPIO DUARTE JÚNIOR
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
27/06/2025 22:32:50.0
ALTAIR OLIVO SANTIN
Avaliador Externo (PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ)

Assinatura Eletrônica
27/06/2025 20:29:09.0
VINÍCIUS FÜLBER GARCIA
Coorientador(a) (UNIVERSIDADE FEDERAL DO PARANÁ)

"Não sou nada.
Nunca serei nada.
Não posso querer ser nada.
À parte isso, tenho em mim todos os sonhos do mundo."
- Fernando Pessoa

AGRADECIMENTOS

Inicialmente, gostaria de agradecer ao Senhor, que é o céu e a terra, por me garantir a chance de aproveitar cada momento. Ainda que eu não mereça, Ele não desviou seu olhar de mim, e mesmo que eu tenha sido ridículo, Ele não me abandonou.

Agradeço também a todos os meus colegas de laboratório, desde os de graduação até professores doutores, passando pelos colegas mestrandos e doutorandos. Especialmente, agradeço aos professores Jorge, Vinícius e João, que foram essenciais para a consolidação do presente trabalho. Também agradeço ao professor Gregio, que tem me orientado desde a graduação.

Estendo os agradecimentos à minha mãe, Ana Cristina, e ao meu pai, André, por terem sido pacientes e compreensivos nos momentos em que eu mais precisei. Por fim, de maneira geral, agradeço a todos os meus amigos e familiares por serem ótimos companheiros.

Obrigado do fundo do coração.

Este trabalho teve apoio da Bluepex CyberSecurity via financiamento de projeto de Inovação da Base Industrial de Defesa – Edital MD/MCTI/FINEP/FNDCT 2022 e da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

RESUMO

Sistemas de Detecção de Intrusão em redes por assinatura proporcionam uma maneira de realizar inspeção de tráfego em um segmento de rede, aumentando assim sua segurança, haja vista a possível detecção de padrões relacionados com atividade maliciosa (assinaturas de *malware*). Devido à execução de *Deep Packet Inspection*, o desempenho desse tipo de solução tende a ser menor à medida que as taxas de transmissão ou a quantidade de assinaturas presentes na base aumentam. Uma possível abordagem para o aumento de desempenho dessas soluções é a implementação em espaço de *kernel*, mais especificamente, em espaço de *driver*, antes que as estruturas que representam um pacote sejam alocadas pelo sistema operacional. Desta forma, é possível escrever um programa BPF (*Berkeley Packet Filter*), anexado à primeira camada da pilha de rede do kernel Linux, o XDP (*eXpress Data Path)*, que tem por objetivo realizar a detecção de assinaturas de *malware*. Este trabalho propõe o Sapo-boi (Sistema de Avaliação e Processamento de tráfegO baseado em BPF/XDP para Observação de Intrusões), a fim de mostrar a viabilidade da implementação de um sistema de detecção de intrusões por assinatura em espaço de *kernel/driver*, bem como discutir as implicações e limitações de fazê-lo. A solução é dividida em dois módulos, o Módulo de Suspeição: um programa BPF/XDP usado para o casamento inicial de padrões maliciosos em espaço de kernel, que redireciona pacotes considerados suspeitos para o espaço de usuário através de *sockets* XDP; e o Módulo de Avaliação: processo de usuário, para onde os pacotes oriundos do Módulo de Suspeição são redirecionados para avaliação aprofundada e veredito final. Os experimentos foram executados de forma a comparar o Sapo-boi com outras três soluções: duas delas totalmente em espaço de usuário, e uma que, como o Sapo-boi, também é dividida entre espaços de *kernel* e usuário. Os resultados apontam uma queda significativa da taxa de pacotes não avaliados pela solução proposta quando comparada às soluções em espaço de usuário, bem como indicam que a maneira como o Sapo-boi realiza a passagem de pacotes para o espaço de usuário é mais robusta do que a maneira realizada pela outra solução em *kernel*. Mais especificamente, num cenário com alto número de assinaturas a serem avaliadas (16 mil) e 10Gbps, o Sapo-boi deixa de avaliar pouco mais de 2% dos pacotes, ao passo Suricata e Snort deixam de avaliar 9,61% e 91,7%, respectivamente. Com relação à outra solução em espaço de kernel, com as mesmas métricas citadas anteriormente, o Sapo-boi é capaz de redirecionar 99,99% dos pacotes ao espaço de usuário, ao passo que a solução mencionada redireciona pouco mais de 91%, o que pode acarretar na ausência de alertas, que representam a real detecção da atividade maliciosa.

Palavras-chave: Detecção de Intrusão; BPF; XDP; Sockets XDP.

ABSTRACT

Signature-based Network Intrusion Detection Systems provide a way to perform network traffic inspection, increasing its security, considering the possible detection of malicious activity or malware signature related patterns. Due to Deep Packet Inspection, the performance of this type of solution tends to be lower as the transmission rates or the number of loaded signatures increase. A possible solution to improve the performance is to implement these solutions in kernel space, more specifically, in driver space, before the structures that represent a packet are allocated by the operating system. With this approach, it is possible to write a BPF (Berkeley Packet Filter) program to perform malware signature detection, hooked to the first layer of the Linux kernel network stack, the XDP (eXpress Data Path). This work proposes Sapo-boi, aiming to demonstrate the viability of implementing a signature-based network intrusion detection system in kernel/driver space, as well as to discuss its implications and limitations. The solution is divided into two modules, the Suspicion Module: an XDP/BPF program used to initially match malicious patterns, redirecting suspected packets to user space using XDP sockets; and the Evaluation Module: a user process, to which the packet that came from the Suspicion Module is forwarded so it can be deeply evaluated, and the final verdict is given. Experiments were executed in order to compare Sapo-boi with three other solutions: two of them are entirely in user space, and one that, like Sapo-boi, is also divided into kernel and user spaces. The results show a significant drop in the non-evaluated packet rate when compared to the solutions entirely in user space. The results also show that the manner in which Sapo-boi redirects the packets from the kernel to the user space is more robust than the manner done by the other kernel solution. More specifically, in a scenario with a high number of signatures to be evaluated (16000) and 10Gbps, Sapo-boi fails to evaluate just over 2% of packets, while Suricata and Snort fail to evaluate 9.61% and 91.7%, respectively. Regarding the other kernel-space solution, with the same metrics mentioned above, Sapo-boi is able to redirect 99.99% of packets to user space, while the aforementioned solution redirects just over 91%, which may result in the absence of alerts, which represent the real detection of malicious activity.

Keywords: Intrusion Detection; BPF; XDP; XDP sockets.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

AC	*Aho Corasick*
AF_XDP	*Address Family - XDP*
BPF	*Berkeley Packet Filter*
BTF	*BPF Type Format*
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
DPI	*Deep Packet Inspection*
DPDK	*Data Plane Development Kit*
FIFO	*First In First Out*
HIDS	*Host Intrusion Detection System*
HIPS	*Host Intrusion Prevention System*
Gbps	*Gigabits* por segundo
IDS	*Intrusion Detection System*
IP	*Internet Protocol*
IRQ	*Interruption ReQuest*
MTU	*Maximum Transmission Unit*
NAT	*Network Address Translation*
NIDS	*Network Intrusion Detection System*
NIPS	*Network Intrusion Prevention System*
PF IDS	***P**er**F** events IDS*
RSS	*Receive Side Scaling*
Sapo-boi	**S**istema de **A**valiação e **P**rocessamento de tráfeg**O** usando **B**PF/XDP para **O**bservação de **I**ntrusões
softirq	*Software Interruption Handler*
SPAN	*Switch Port Analyser*
TCP	*Transmission Control Protocol*
TC	*Traffic Control*
UDP	*User Datagram Protocol*
UMEM	*User Mode Memory*
VLAN	*Virtual Local Area Network*
VM	*Virtual Machine*
XDP	*eXpress Data Path*
XSK	*XDP socket*

SUMÁRIO

1 INTRODUÇÃO

Sistemas de detecção de intrusão (IDS - *Intrusion Detection Systems*) em redes têm por objetivo avaliar tráfego de rede em busca de padrões de ataque sabidamente conhecidos. Tais soluções se tornam cada vez mais importantes quando avaliam-se os prejuízos causados por ataques, que são cada vez maiores.

De acordo com pesquisa realizada a pedido do Senado Federal, 24% dos brasileiros sofreram um golpe digital (perderam dinheiro devido a fraudes, clonagem de cartão ou invasão de contas bancárias) entre junho de 2023 e junho de 2024 Senado (2024). Quando consideram-se entidades organizacionais (no Brasil), prejuízos variam entre 3 e 5 milhões de reais por ano CNN (2024).

Já pesquisas em nível mundial corroboram com o crescente número de ataques. Somente a empresa de cibersegurança *Avast* reportou que bloqueou 10 bilhões de tentativas de ataque em 2023, o que representa um aumento de 50% em comparação com o ano anterior Avast (2024).

Soluções IDS podem ser usadas para detecção de ataques. Existem dois tipos de implementação para esse tipo de solução. Os IDS baseados em assinatura têm por objetivo avaliar um objeto de interesse (pacote de rede, linha de *cache*, memória de um processo, etc.) a fim de encontrar algum padrão sabidamente malicioso, que represente algum ataque conhecido.

Já os IDS com base em anomalias são soluções que têm por finalidade traçar um padrão de comportamento do objeto de interesse (usuário, atividade dos processos, tráfego de rede, etc). Feito isso, o sistema busca por tudo aquilo que não corresponde àquele comportamento, ou seja, que represente uma anomalia.

Visto isso, tem-se que as soluções IDS por assinatura são responsáveis por identificar padrões com o auxílio de uma base de assinaturas maliciosas. Em outras palavras, o sistema deve comparar o objeto com algum padrão malicioso que esteja na base. Caso haja correspondência de padrões, aquele objeto é rotulado como malicioso/suspeito.

Já para as soluções baseadas em anomalia, a base não contém padrões maliciosos, mas padrões benignos. Todo objeto que não se enquadre nos padrões rotulados como benignos é considerado malicioso.

Ambos os tipos de implementação possuem problemas. As soluções baseadas em anomalia apresentam um alto número de falsos positivos Jyothsna et al. (2011). Isso pode ser explicado pelo fato de ser difícil determinar o que deve ser considerado como um comportamento benigno, haja vista que tudo que não está neste escopo será considerado como uma intrusão.

O principal problema das soluções baseadas em assinatura é a limitação da base Hubballi e Suryanarayanan (2014). Mesmo que hajam milhares de assinaturas, este tipo de sistema não é capaz de detectar ameaças desconhecidas, nem qualquer ameaça cuja assinatura não esteja presente. Outro problema é a própria quantidade de assinaturas, que é proporcional ao tempo que o sistema leva para avaliar um único objeto. Portanto, quanto mais assinaturas, menor é o desempenho do sistema Firoz et al. (2020).

Porém, ambas as abordagens também possuem vantagens. As soluções por anomalia são capazes de detectar ameaças ainda desconhecidas pelo sistema, ao passo que as soluções por assinatura são mais assertivas, ou seja, determinam fielmente qual foi o *malware* avaliado, permitindo a análise da assinatura maliciosa.

A presente dissertação tem por escopo os IDSs de rede (NIDS - *Network IDS*) baseados em assinatura. Esse tipo de solução tem por finalidade encontrar padrões maliciosos em pacotes de rede. NIDS vêm sendo usados desde 1998, quando o utilitário *tcpdump* foi reestruturado, com

o objetivo de criar o primeiro NIDS: o Snort[1]. Também deve-se destacar o Suricata[2], sistema introduzido no mercado como um concorrente *multi thread* do Snort.

Mesmo as implementações consolidadas, como as descritas acima, sofrem com problemas de desempenho. Num primeiro momento, a solução deve realizar a aquisição de pacotes, para o posterior processamento (busca por padrões maliciosos). Note que tais soluções são implementadas como aplicações em espaço de usuário, que devem realizar a coleta de pacotes assim que eles são recebidos, ainda em espaço de kernel.

Todo pacote ingressante passa pela pilha de rede do sistema operacional, que está em espaço de *kernel*. Para que a solução seja capaz de capturar os pacotes para análise, ela deve determinar como os pacotes serão coletados.

O Snort, por padrão, realiza a aquisição de pacotes através da biblioteca *libdaq*[3], que provê uma camada de abstração para a execução da clássica biblioteca *libpcap*. O Suricata, por sua vez, realiza essa ação através de *sockets AF_PACKET*. Essas ferramentas são responsáveis por criar cópias dos pacotes e enviá-las ao espaço de usuário, onde a detecção de padrões ocorre.

Note que, quanto maior a taxa de recebimento de tráfego, maior o uso de processamento num contexto de aquisição de pacotes, ou seja, em espaço de kernel. Neste caso, o uso de CPU deve ser escalonado entre o processamento da pilha de rede (atividade normal para o tratamento de pacotes ingressantes) e a aquisição de pacotes (atividade da solução), o que pode prejudicar o desempenho de ambas as tarefas Hu et al. (2020).

Uma vez que os pacotes são capturados, eles são enviados para o espaço de usuário para o casamento de padrões, bem como outras atividades configuráveis que o sistema IDS pode realizar. Dependendo da quantidade de padrões que o IDS deve buscar e da presença ou não de expressões regulares, o tempo de CPU gasto nessa etapa também pode ser elevado. Por fim, caso algum padrão malicioso seja encontrado, o sistema deve tomar uma ação. Geralmente, essa ação é uma escrita em um arquivo de log, indicando que houve coincidência com alguma assinatura de *malware* carregada.

O objetivo deste trabalho é abordar uma alternativa ao modelo de aquisição e processamento descrito acima, comparando a solução desenvolvida com Snort e Suricata. A proposta consiste em realizar o processamento inicial dos pacotes ainda em espaço de *kernel*, através das tecnologias BPF (*Berkeley Packet Filter*) e XDP (*eXpress Data Path*). Se esta etapa indicar que o pacote sendo processado é considerado suspeito, ele deve ser redirecionado para espaço de usuário, que dará o veredito final para aquele pacote, escrevendo nos arquivos de *log* caso haja a confirmação da presença de alguma assinatura de *malware*. Em outras palavras, num primeiro momento, ainda em espaço de *kernel*, um programa BPF pode levantar suspeita da presença de uma assinatura em um pacote. O pacote é então redirecionado para processamento por um processo de usuário, que pode confirmar ou não a presença de atividade maliciosa.

1.1 PROPOSTA

Levando em consideração o objetivo de usar uma implementação diferente dos demais sistemas, esta Seção tem por objetivo definir uma nova solução, identificando seus componentes. A solução proposta neste trabalho será referenciada como **Sapo-boi** (Sistema de Avaliação e Processamento de tráfegO usando BPF e XDP para Observação de Intrusões), definida inicialmente pelo autor desta dissertação em Machnicki et al. (2024).

[1] https://www.snort.org/snort3

[2] https://suricata.io/

[3] https://github.com/snort3/libdaq

O sistema proposto faz uso do XDP, um *hook* BPF na pilha de rede do *kernel* Linux capaz de processar pacotes (através de programas BPF) antes mesmo do sistema operacional alocar as estruturas para representá-los. Um programa BPF anexado ao *hook* XDP opera no momento mais anterior possível em que existe a representação de um pacote: logo após a transferência dos *bytes* do mesmo para o *ring buffer* do sistema operacional, onde os pacotes ingressantes são temporariamente armazenados, até o processamento da pilha de rede do kernel.

Portanto, um programa BPF anexado ao *hook* XDP representa o ponto mais antecipado possível pela qual um usuário pode definir rotinas de processamento para pacotes de rede. Este tipo de programa pode executar paralelamente, uma instância por núcleo de CPU.

O Sapo-boi usa BPF/XDP para a detecção antecipada de padrões maliciosos em pacotes de rede, através do algoritmo de detecção de padrões Aho-Corasick Aho e Corasick (1975). A solução ainda faz uso de *sockets XDP* para redirecionar pacotes considerados suspeitos, ou seja, cujo *payload* contém parte de uma assinatura maliciosa, do espaço de *kernel* para o espaço de usuário.

Sockets XDP representam uma família que tem por objetivo permitir que os pacotes contornem a pilha de rede do *kernel* Linux, evitando todo o processamento complexo que nela ocorre. Diferentemente dos *sockets AF_PACKET*, usados pelo Suricata, não são realizadas cópias de pacotes.

O Sapo-boi é dividido em dois módulos: um em espaço de usuário, o Módulo de Avaliação; e outro em espaço de *kernel*, o Módulo de Suspeição. Este último é capaz de detectar padrões maliciosos em paralelo, logo na primeira camada da pilha de rede (o XDP), usando autômatos Aho-Corasick, e enviar pacotes considerados suspeitos para o espaço de usuário através de *sockets* XDP. O Módulo de Suspeição ainda é capaz de alocar espaço para metadados e enviá-los junto com os pacotes, permitindo que dados calculados/coletados em espaço de *kernel* sejam enviados para o espaço de usuário.

Já o Módulo de Avaliação é a aplicação de usuário que receberá os pacotes suspeitos. Ele é capaz de avaliar os metadados enviados pelo Módulo de Suspeição a fim de determinar de maneira eficiente quais padrões ainda devem ser avaliados para confirmar a suspeita levantada em espaço de kernel, além de escrever nos arquivos de *log* caso a suspeita se confirme verdadeira. Além disso, o Módulo de Avaliação também é responsável pela inicialização de estruturas que permitem o correto funcionamento do programa BPF (Módulo de Suspeição).

1.2 DIVISÃO DA ARQUITETURA DE DETECÇÃO

A possibilidade de implementar um NIDS usando XDP é exposta inicialmente em Wang e Chang (2022). O trabalho é a base da presente dissertação, evidenciando a divisão da detecção de padrões entre os espaços de kernel e de usuário. Os autores mostram ainda como realizar a avaliação dos pacotes para a busca de padrões maliciosos. Porém, é válido evidenciar que o trabalho tem problemas de estrutura e aplicabilidade. Particularmente, o trabalho diz operar como uma solução IDS, ou seja, que opera com cópias de pacotes, porém realizam os testes comparando a solução desenvolvida com o Snort, que deve por sua vez realizar a cópia dos pacotes para a análise. Ou seja, a análise compara uma solução que deve realizar cópias (o Snort) com a solução desenvolvida, que não o faz, sem implementar mecanismos que tornem a comparação justa.

Neste trabalho, o Snort também foi objeto de comparação, porém, mecanismos para a equidade dos experimentos foram implantados. Mais especificamente, uma função de cópia foi adicionada ao kernel do sistema operacional do *host* que estava executando a solução desenvolvida, garantindo a cópia de todos os pacotes, mesma abordagem feita pelo Snort e Suricata. Ainda

no campo dos experimentos, Wang e Chang (2022) comparam a solução que desenvolveram com uma versão *single thread* do Snort, arquitetura obsoleta desde 2021, e não foi comparada com o Suricata, que é sabidamente o estado da arte com relação às soluções NIDS que operam processando pacotes em espaço de usuário Machnicki et al. (2024).

1.3 CONTRIBUIÇÕES

Este trabalho apresenta as seguintes contribuições principais:

- Demonstração da viabilidade e a maneira de implementar a arquitetura exibida, que conta com mecanismos de detecção de padrões em espaço de kernel, usando BPF e XDP, por meio da comparação com soluções estado da arte que não o fazem;
- Projeto e prototipação de um NIDS robusto com processamento e redirecionamento de pacotes por meio de *sockets* XDP, enviando metadados juntamente com o pacote para o espaço de usuário para auxiliar no processo de detecção de ameaças;
- Comparação de desempenho computacional da solução proposta com outros NIDS já existentes: dois deles executando inteiramente em espaço de usuário (Snort e Suricata) e uma, que assim como o Sapo-boi, é mista, em espaços de *kernel* e usuário (descrita em Wang e Chang (2022)).

Os experimentos realizados demonstram que o Sapo-boi supera as soluções inteiramente em espaço de usuário na capacidade de processamento de tráfego, permitindo que mais pacotes sejam avaliados. Quando comparado com a solução mista, nota-se a capacidade superior que a solução proposta tem de enviar pacotes suspeitos ao espaço de usuário, o que permite que mais pacotes sejam analisados neste contexto, proporcionando que mais alertas indicando assinaturas maliciosas sejam gerados.

1.4 ORGANIZAÇÃO DO TRABALHO

O restante deste artigo está organizado da seguinte forma: o Capítulo 2 apresenta conceitos fundamentais para a compreensão do trabalho; o Capítulo 3 exibe os principais trabalhos relacionados e os classifica em trabalhos que comparam ou propõem soluções IDS; o Capítulo 4 descreve a arquitetura e a implementação da solução proposta; o Capítulo 5 descreve a infraestrutura usada para os testes, detalhando cada componente que a compõe; o Capítulo 6 apresenta e discute os resultados experimentais, comparando-os com o estado da arte; e, finalmente, o Capítulo 7 apresenta as considerações finais e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os fundamentos teóricos deste trabalho. Inicialmente, serão apresentados os conceitos relacionados à detecção de intrusão, incluindo conceitos e técnicas usados por soluções IDS para sua execução, como, por exemplo, os *fast patterns*, que são os primeiros padrões que as soluções de IDS por assinatura irão buscar durante o processo de detecção. Em seguida, as tecnologias utilizadas para o desenvolvimento do trabalho serão apresentadas em detalhe, especialmente BPF, XDP e *sockets AF_XDP*. O algoritmo Aho-Corasick também será explicado em profundidade, bem como os conceitos de *softirq* e *perf events*, estruturas específicas do sistema operacional Linux.

2.1 DETECÇÃO DE INTRUSÃO

Detecção de intrusão é o processo de monitoração de sistemas em busca de evidências de ataques via inspeção e análise de eventos Liao et al. (2013). NIDS são soluções de análise de tráfego de rede para identificar ameaças e gerar alertas, seja por anomalia (detecção de comportamentos) ou casamento de assinaturas. Limitando o escopo para detecção por assinatura, os NIDS realizam DPI (*Deep Packet Inspection*) no tráfego de rede em busca de padrões suspeitos previamente definidos em conjuntos de regras Lin et al. (2008). Cada regra pode ser vista como uma assinatura de *malware* que, se identificada nos pacotes, representa um gatilho para que a solução gere um alerta. Cada regra pode conter um ou mais padrões maliciosos. Para que o alerta seja gerado, todos eles devem estar presentes no tráfego avaliado.

2.1.1 Posicionamento de Sistemas de Detecção de Intrusão

Um NIDS pode ser posicionado de duas maneiras na infraestrutura de rede. Como indicado na Figura 2.1, quando configurada para o modo de uso padrão, o NIDS recebe uma cópia dos pacotes. Caso o conteúdo do tráfego analisado coincida com alguma assinatura de *malware* presente nas regras carregadas pela solução, a mesma pode informar o evento aos administradores da rede, por meio de alertas, escritos em arquivos de *log*. Nota-se que o *switch* deve ser configurado de maneira a realizar o espelhamento de tráfego, através de uma técnica conhecida como *SPAN (Switch Port ANalyser)*.

A Figura 2.1 também mostra que os NIDS recebem como entrada um arquivo de configuração e um arquivo de regras. O arquivo de configuração permite definir variáveis e comportamentos que o sistema vai adotar, como por exemplo definir qual é o endereço da rede local ou instruir a sistema a não realizar DPI em pacotes cifrados.

Já o arquivo de regras tem por função definir as assinaturas de ataque, ou seja, quais padrões o sistema deve detectar, por meio de DPI, para indicar em seus *logs* que houve uma tentativa de intrusão. A Subseção 2.1.2 detalha como são construídas as regras de um IDS por assinatura.

Existem vantagens e desvantagens em posicionar o sistema dessa maneira. Como principal vantagem, destaca-se o fato da solução não comprometer o desempenho da infraestrutura: ainda que os processos de aquisição de pacotes e DPI sejam custosos, não há influência na taxa de recebimento de pacotes pela rede interna, haja vista que a solução está analisando cópias.

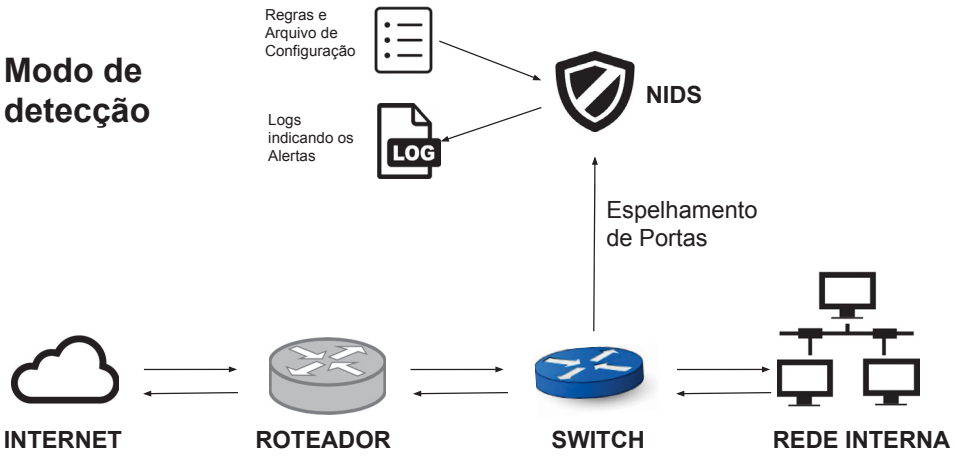


Figura 2.1: NIDS com tráfego espelhado.

Como principal desvantagem, está o fato de que mesmo que o sistema considere que no tráfego existam assinaturas de *malware*, o tráfego original já foi recebido pela rede interna. Em outras palavras, quando o sistema detecta a intrusão, ela já está em progresso.

Existe também a possibilidade de posicionar o sistema entre o *switch* e a rede. Nesse caso, tem-se um NIPS, ou *Network Intrusion Prevention System*. A Figura 2.2 mostra a arquitetura do sistema no modo de prevenção.

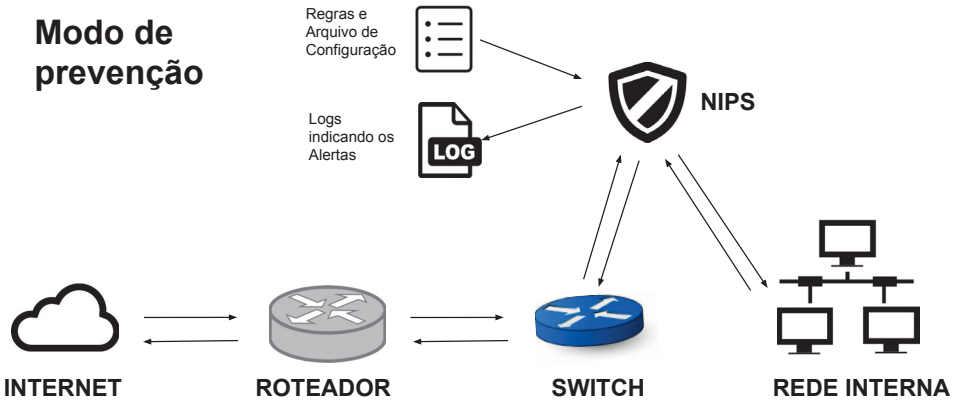


Figura 2.2: NIPS (Modo *Inline*).

Neste modo, o sistema é bloqueante, porque não permite que o pacote siga para a rede interna antes de terminar de realizar DPI. Apesar desse modo ser bastante útil, por permitir que o sistema não somente gere alertas, mas de fato bloqueie os pacotes e encerre conexões, ele não é largamente usado, porque limita a vazão dos pacotes pela rede, criando um gargalo na arquitetura.

Também é possível posicionar um sistema que analisa tráfego de rede nos nós da rede interna. Neste caso, o sistema receberá a denominação de HIDS ou HIPS (*Host IDS/IPS*), dependendo se somente irá detectar ameaças (HIDS), ou se de fato tomará alguma medida para descartar os pacotes maliciosos (HIPS). A principal vantagem de um IDS de *host* é que a análise pode ser feita em conjunto com outros indicativos ocorrendo no sistema, por exemplo, um HIDS/HIPS pode fazer instrumentação de funções (de kernel, usuário ou *syscalls*) em busca de algum indicativo de ataque. Como desvantagem, cita-se que esse tipo de solução pode onerar

o desempenho do *host*, que contará com uma aplicação de processamento exaustivo competindo por escalonamento com as demais tarefas do sistema.

2.1.2 Regras de Sistemas de Detecção de Intrusão

As regras de um IDS representam assinaturas de *malware*. Cada regra é composta por um conjunto de padrões. Quando combinados, esses padrões representam a impressão digital de um *malware* específico.

A Figura 2.3 ilustra uma regra simples suportada por Snort e Suricata (ambos NIDS por assinatura). Antes do "->", mostra-se a ação a ser tomada no caso de ativação (neste caso, gerar um alerta), bem como o protocolo a ser procurado nos cabeçalhos do pacote avaliado (TCP), o endereço IP (qualquer - *any*) e a porta de origem (13, neste caso). Após o "->", há o endereço IP e a porta de destino do tráfego analisado, que na Figura 2.3 são representados, respectivamente, pela rede 192.168.1.0/24 e pela porta 99.

alert tcp any 13 -> 192.168.1.0/24 99 (content: "padrão"; fast_pattern; content:"outroPadrão"; sid:1)

Figura 2.3: Exemplo de regra de IDS.

Entre parênteses, estão as opções da regra. Elas contêm, entre outros campos possíveis, os padrões a serem buscados no tráfego de rede (*content*, pode existir zero ou múltiplos); um padrão de reconhecimento rápido que será o primeiro testado, por ser o mais representativo da assinatura (*fast_pattern*, pode existir zero ou um, definido pelo *content* imediatamente antes da palavra-chave) e um identificador único da regra (*sid*, exatamente um).

No caso da regra apresentada na Figura 2.3, os *contents* são representados como *strings*, mas também existe a possibilidade de representá-los como uma sequência de *bytes*.

O gatilho para executar a ação da regra é que todos os padrões nela contidos estejam presentes no pacote. Caso uma das regras carregadas gere um alerta, o sistema para de analisar o pacote corrente. Em outras palavras, para cada pacote, o sistema tentará cada regra carregada, até que uma delas funcione. As regras carregadas são avaliadas de cima pra baixo. Caso uma delas funcione, nenhuma outra será avaliada. Portanto, as regras em si formam um OU lógico.

Como todos os padrões devem estar no pacote, pode-se dizer que os padrões de uma regra formam um E lógico. O sistema avalia as regras da esquerda para a direita; caso uma opção não esteja presente, as demais não serão avaliadas.

2.1.3 *Fast Patterns*

No exemplo mostrado na Figura 2.3, o *content* "padrão"é um *fast pattern*. Isso significa que o conjunto de bytes que formam a *string* "padrão"será o primeiro que o sistema procurará no *payload* dos pacotes avaliados.

Os *fast patterns* servem como mecanismo para melhoria de desempenho. Se o *fast pattern* da regra não estiver presente no pacote, significa que aquele pacote não representa ameaça, quando levada em consideração a assinatura definida pela regra. Em outras palavras, se o *fast pattern* de uma regra não estiver presente no pacote, o processamento da mesma deve ser encerrado (note que todos os *contents* de uma regra devem estar presentes para que o sistema execute alguma ação, como o *fast pattern*, que é um *content*, não foi encontrado, o sistema pode avaliar a próxima regra).

2.2 BPF

O BPF (*Berkeley Packet Filter*) McCanne e Jacobson (1993) foi inicialmente desenvolvido como uma solução para filtragem de pacotes no kernel do Linux, que possui suporte à tecnologia desde a versão 2.5. O utilitário *tcpdump*[1], através da biblioteca *libpcap*, é o usuário mais conhecido do BPF clássico (cBPF), como é chamada a tecnologia quando pretende-se referenciar suas primeiras versões. Mais especificamente, os filtros do *tcpdump* são reduzidos a programas BPF, que verificam os pacotes em busca das expressões definidas.

O BPF conta com dois componentes principais: uma máquina virtual (VM) em espaço de kernel e um conjunto de instruções *bytecode*, que serão executadas por esta VM. Para obter o código *bytecode* BPF, um usuário pode escrever um programa em C, que deve ser compilado para se tornar um arquivo objeto, que contém o código de *bytes* suportado pela VM.

A escrita de programas BPF (ou melhor, programas C que serão posteriormente traduzidos para *bytecode* BPF) deve ser bastante cautelosa. Além de checar cada um dos acessos à memória, os programas devem atender a algumas restrições. A maior delas é o fato de que não há *heap* em programas BPF. Portanto, não é possível alocar memória dinamicamente. Outro fator limitante é a quantidade de instruções *bytecode* possíveis em um único programa BPF, que é de 1 milhão.

A Figura 2.4 mostra como os programas BPF são carregados para o kernel. O programa escrito em C deve ser compilado pelo LLVM/clang no modo *cross compiler*, especificando o alvo para *bytecode* BPF, que estará contido num arquivo objeto. Atualmente, este é o único compilador capaz de realizar esta tarefa.

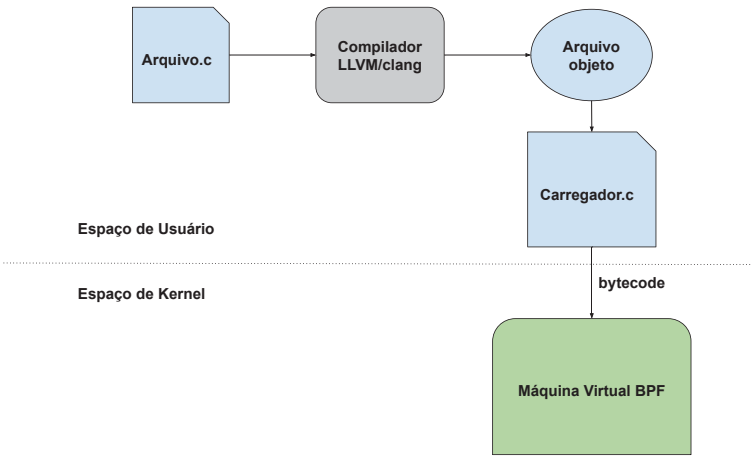


Figura 2.4: Carregamento de um programa BPF.

Uma vez que o compilador gera o objeto, é possível carregá-lo para espaço de kernel. Este processo pode ser realizado através de um carregador próprio (escrito pelo usuário), que lê o arquivo objeto e invoca as chamadas de sistema (*syscalls*) específicas de carregamento de programas BPF.

Antes do programa ser carregado para a VM BPF em kernel, ele passa por uma série de rotinas responsáveis por tratar das verificações impostas pela arquitetura BPF. Uma vez que o *bytecode* é carregado para a VM, esta será responsável por realizar a tradução para instruções nativas da máquina *host*, num processo de compilação *Just In Time*.

Um dos aspectos fundamentais, e que garante a segurança de programas BPF, é a presença do verificador de código estático, chamado comumente de verificador BPF. Depois que

[1] https://www.tcpdump.org/

um programa é compilado para *bytecode* BPF, o código gerado é verificado para garantir que não haja acessos não verificados à memória, *loops* sem número pré-definido de iterações, e que o programa termine. Para garantir a ausência de *loops* infinitos e a finitude do programa, deve ser possível representá-lo através de um grafo direcionado acíclico, em que cada nó é composto por um conjunto de instruções *bytecode* SANTOS et al. (2019). Se a construção de tal estrutura for inviável, há uma falha de carregamento.

2.2.1 Mapas BPF

Com o passar do tempo e das atualizações de kernel, novas funcionalidades foram adicionadas ao BPF. Como destaque, deve-se mencionar o suporte a mapas BPF, que podem ser definidos como áreas de memória compartilhadas que permitem comunicações entre programas BPF e entre programas BPF e processos de usuário.

Como não é possível alocar memória dinamicamente, somente na *stack*, todos os dados que o programa BPF necessita para executar devem ser pré-preenchidos nos mapas. Apesar de aumentar a segurança, essa abordagem limita bastante a viabilidade de execução de programas suficientemente genéricos na VM BPF.

Os mapas são estruturas de dados chave-valor localizadas em espaço de kernel. Essas estruturas devem ser definidas no mesmo arquivo (escrito em C) que contém a lógica de execução do programa BPF, porém, em uma seção específica. Portanto, o *bytecode* gerado pelo compilador LLVM possui as definições dos mapas e contém metadados que indicam em que parte do objeto eles estão definidos.

Quando o carregador verifica que existem definições de mapas no *bytecode*, ele chama uma *syscall* específica para a criação dos mapas em espaço de kernel. Os mapas serão concebidos com sucesso somente se não houver falhas nas rotinas de verificação do BPF.

Uma vez que os mapas são criados, cabe a um processo de usuário populá-los. Tal ação pode ser realizada com auxílio das funções *bpf_object__find_map_by_name*, que é capaz de retornar um descritor para o mapa, recebendo como parâmetros o arquivo objeto e o nome do mapa; e *bpf_map_update_elem*, que recebe o descritor do mapa, bem como os valores para o par chave-valor que irão compor uma entrada do mesmo.

A Figura 2.5 sumariza o processo de criação e preenchimento dos mapas. Num primeiro momento, o próprio carregador verifica que foram definidos mapas, pela presença de seções específicas no arquivo objeto. A seguir, ele chama a *syscall* BPF, que cria os mapas, em espaço de kernel.

Ademais, um processo de usuário já pode acessar os mapas recém-criados, preenchendo-os. Por fim, o processo ainda pode definir rotinas de consulta e atualizações aos mapas. Note que tanto o programa BPF quanto o processo de usuário têm acesso aos mapas, portanto, ambos podem atualizá-los.

Atualmente existem 33 tipos diferentes de mapas BPF. Alguns deles representam estruturas de dados genéricas, como por exemplo vetores e tabelas *hash*. Existem também mapas específicos, por exemplo, mapas para o envio de eventos do programa em espaço de kernel (programa BPF) para processos de usuário.

Um exemplo do uso dos mapas é a verificação de portas TCP específicas. Caso o objetivo de um programa BPF seja verificar a presença de uma lista de pares de portas, um mapa do tipo *hash* pode ser preenchido de maneira que as chaves representem a *hash* de um par de portas específico.

Quando o programa BPF recebe um pacote para processamento, ele pode verificar se este tem como protocolo de transporte o TCP, e em caso positivo, verificar quais são as portas definidas no cabeçalho. Visto isso, o programa pode preencher uma estrutura que é

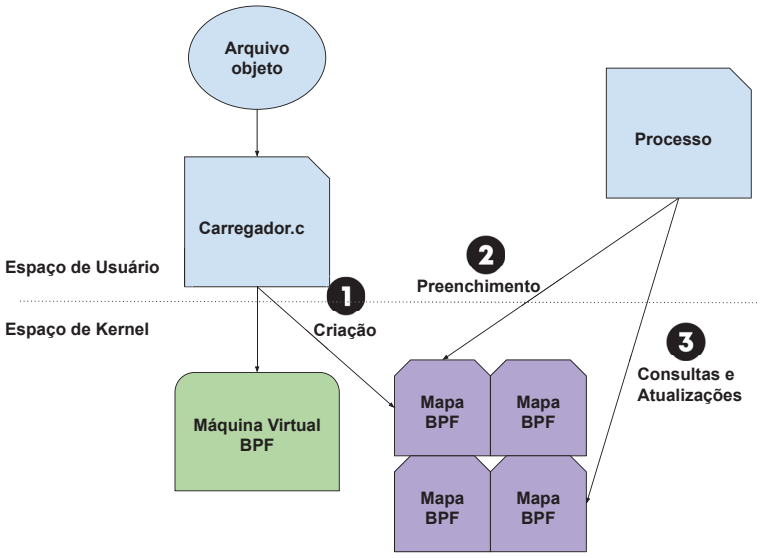


Figura 2.5: Carregamento mapas BPF.

composta por dois inteiros de 16 *bits*, que representam o par de portas, e chamar a função *bpf_map_lookup_elem*, passando como parâmetro o mapa, que está definido no mesmo arquivo, e a chave, que é a estrutura recém-preenchida. Se alguma entrada do mapa for a representação do mesmo par de portas encontrado no pacote, a função indicará a qual entrada do mapa aquele par de portas está associado.

Com o advento dos mapas, o aumento do tamanho dos registradores da VM BPF e a introdução de *tail calls* (funções que invocam outros programas BPF, devido a limitações no número de instruções), o BPF passou a ser conhecido como eBPF, ou *extended* BPF.

Programas BPF só podem ser adicionados em regiões do kernel que deem suporte à sua execução. Em mais detalhes, existem regiões de kernel que funcionam como *hooks* para programas BPF. Cada vez que algum evento de interesse ocorre nessa região específica, um programa BPF pode ser executado. Como exemplo, cita-se o *hook* TC, ou **trafic control**. O TC é uma região de execução localizada na pilha de rede do kernel Linux. Cada pacote recebido ou enviado pelo sistema é gatilho para a execução de um programa BPF ligado ao TC.

Também é importante mencionar que o BPF é uma tecnologia intrinsecamente paralela. Ou seja, cada núcleo de CPU executa uma instância própria do programa. Mapas BPF podem ser compartilhados ou não entre os programas de cada núcleo, dependendo de seu tipo. Por exemplo, mapas do tipo *BPF_MAP_TYPE_ARRAY* são compartilhados com todas as instâncias, ao passo que mapas do tipo *BPF_MAP_TYPE_PERCPU_ARRAY* são únicos para cada uma delas.

2.3 XDP

O XDP (*eXpress Data Path*) pode ser definido como um *hook* BPF. Mais especificamente, o XDP é a primeira camada da pilha de rede do kernel Linux, assim como demonstrado na Figura 2.6.

2.3.1 Pilha de Rede do Kernel Linux

Além do XDP, a pilha de rede é composta pela camada TC, que permite a configuração do controle de tráfego. Mais especificamente, o TC permite modelar o tráfego egresso, garantindo melhor comportamento da rede, além de escalonar o tráfego a ser enviado. Além disso, é possível

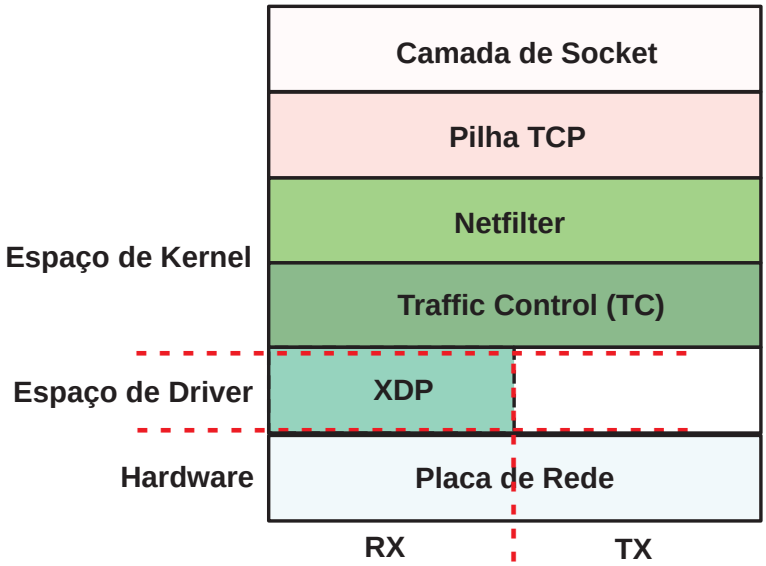


Figura 2.6: Pilha de Rede do Kernel do Linux. Adaptado de SANTOS et al. (2019).

determinar políticas de classificação para o tráfego recebido, bem como descartar pacotes que não obedecem a essas políticas Hubert (2025).

É possível definir programas BPF que funcionam como filtros de classificação, e anexá-los na camada/*hook* TC. O *netfilter*[2] é a camada responsável pela filtragem de tráfego desde o kernel 2.4. Os *firewalls* intrínsecos ao sistema operacional Linux, *iptables* e *nftables*, são localizados nesta camada..

A pilha TCP pode ser vista como a camada responsável pela interpretação e resposta às *flags* TCP, enquanto a camada de *socket* é responsável pela interação com as aplicações.

Cada um dos pacotes recebidos ou enviados passa por todas as camadas da pilha de rede, com a única exceção sendo a camada XDP, que só processa pacotes ingressantes.

2.3.2 Programas XDP

Um programa BPF pode ser acoplado ao *hook* XDP, sendo assim chamado de programa XDP. Este tipo de programa é executado a cada vez que um pacote é recebido. Mais especificamente, o programa XDP é posto em execução ainda em espaço de *driver*, como é possível verificar na Figura 2.6, logo após o DMA (*Direct Memory Access*) de rede.

Mais especificamente, após o DMA, ou seja, a transferência, realizada pela placa de rede, do pacote recebido para uma região de memória específica, neste caso, o *ring buffer* de rede do kernel linux, o *driver* da placa de rede entra em ação para realizar o tratamento do pacote recebido. Em dado momento, o *driver* verifica se há um programa XDP vinculado àquela interface. Se houver, o *driver* invoca a execução do mesmo, passando como parâmetro um contexto, definido na Listagem 2.1.

Listing 2.1: Estrutura do parâmetro enviado a um programa XDP.

```
1  struct xdp_md {
2      // (Note: type __u32 is NOT the real-type)
3      __u32 data;
4      __u32 data_end;
5      __u32 data_meta;
6      // (Note: type __u32 is the real-type)
```

[2]https://www.netfilter.org/

```
7         __u32 ingress_ifindex;
8         __u32 rx_queue_index;
9     };
```

Apesar de ser representado como inteiros sem sinal de 32 bits (*__u32*), os membros *data* e *data_end* são, respectivamente, o ponteiro para o início e para o final do pacote. Quando o programa XDP deseja acessá-los, deve fazer uma conversão de tipo para *void* *. O campo *data_meta* também deve ser considerado como um ponteiro. Se o programa XDP alocar metadados para serem tratados junto com o pacote, ele pode modificar este campo.

Já o campo *ingress_ifindex* representa em qual o índice da placa de rede na qual o pacote ingressou, enquanto o campo *rx_queue_index* representa o índice da fila desta placa de rede que recebeu o pacote. Ambos os valores podem ser interpretados como inteiros.

A Figura 2.6 ainda mostra outros dois aspectos do XDP. O primeiro deles é que o gatilho para execução é o recebimento de tráfego, ou seja, programas XDP não são executados para pacotes enviados. O outro ponto é que a camada XDP é a primeira camada de *software* da pilha de rede. Esse fato indica que programas XDP representam o momento mais anterior possível que um usuário pode realizar processamento de tráfego recebido (ainda em espaço de *driver*).

2.3.3 Tipos de retorno e modos de operação

Um programa XDP deve retornar uma ação. Uma ação pode ser vista como o veredito para o pacote sendo processado. Existem cinco tipos de retornos disponíveis:

- XDP_ABORTED: indica que houve um erro de processamento. Neste caso, o pacote é descartado e a exceção *trace_xdp_exception* é levantada. É possível verificar mais detalhes sobre o erro através do arquivo */sys/kernel/debug/tracing/events/xdp/xdp_exception/*, que contém os *logs*;
- XDP_DROP: o pacote é descartado silenciosamente, ou seja, o veredito para o pacote corrente é de descarte, sem erros de processamento;
- XDP_PASS: permite que o pacote continue até o kernel, passando pelas demais camadas da pilha de rede;
- XDP_TX: retransmite o pacote pela mesma interface de rede em que chegou. É útil quando alguns campos do pacote são modificados, e o mesmo pode ser retransmitido;
- XDP_REDIRECT: Redireciona o pacote. Existem três tipos de redirecionamento: para outra interface de rede (física ou virtual), para outro núcleo de CPU, ou para um programa de usuário.

Ao contrário das demais, a ação *XDP_REDIRECT* não deve ser retornada sozinha, mas através de funções auxiliares específicas para este propósito: *bpf_redirect() e bpf_redirect_map()*. A primeira serve para o redirecionamento do pacote corrente para outra interface. A segunda usa mapas BPF para redirecionar o pacote. Para redirecionamento para outra CPU, um mapa do tipo *BPF_MAP_TYPE_CPUMAP* deve ser empregado. Como mostrado na Seção 2.2, mapas são estruturas de dados chave-valor. Neste caso, é possível definir a chave como um número inteiro, representando qual CPU deve processar o pacote, enquanto o valor é uma referência para a própria CPU destino.

Para o redirecionamento para espaço de usuário, são usados mapas do tipo *BPF_MAP_TYPE_XSKMAP*, nos quais as chaves do mapa são números inteiros, que representam

os índices das filas da placa de rede, enquanto os valores são descritores para *sockets* XDP, estruturas específicas para realizar o redirecionamento de pacotes do espaço de kernel para usuário.

O número de entradas criadas neste mapa deve ser igual (ou menor) ao número de filas da placa de rede, haja vista que pode existir um socket por fila, e cada socket tem o papel de redirecionar os pacotes que ingressaram pela fila à qual está vinculado para espaço de usuário. Mais especificamente, o *socket* vinculado ao índice *i* do mapa será responsável por redirecionar cada pacote que ingressar através da fila *i* da placa de rede.

Existem três modos de operação para programas XDP: simulado, nativo e *offload*. O modo simulado indica que o *driver* da placa de rede não dá suporte ao carregamento de programas XDP. Na prática, isso significa que o programa não executará em espaço de *driver*, mas em espaço de kernel. Com isso, algumas estruturas auxiliares serão alocadas, mais notadamente o preenchimento de um *sk_buff*, estrutura que representa um pacote em espaço de kernel The Kernel Development Community (2025). Caso o *driver* mantivesse suporte, a execução do programa se daria antes dessa operação.

No modo nativo, o programa XDP executa verdadeiramente em espaço de *driver*, sem a necessidade de instâncias de estruturas auxiliares. Para isso, é necessário que o *driver* da placa forneça suporte ao XDP [3]. Já no modo *offload*, o programa é carregado para uma placa de rede, que deve possuir suporte para a execução de programas BPF. Atualmente, as únicas placas de redes com suporte à execução de programas XDP são da linha *agilio*, da marca *Netronome* [4]. Este modo é o mais adequado para redes de alta velocidade, haja vista que, como o programa executa diretamente na placa de rede, nenhum ciclo de CPU do *host* é usado para o processamento dos pacotes.

2.4 SOCKETS XDP

Um único programa BPF/XDP é limitado quanto à quantidade de instruções que pode executar. Se for necessário mais processamento, o pacote deve passar pela pilha de rede até alcançar um programa no espaço do usuário, ou ser redirecionado para outro programa BPF, através da técnica de *tail call*, que permite que um programa BPF chame outro, com o mesmo contexto (para o caso de programas XDP, o mesmo pacote).

Sockets da família XDP (AF_XDP ou XSK) são projetados para redirecionar pacotes recebidos do kernel para o espaço do usuário, que não tem limitações quanto ao número de instruções. Nesse sentido, eles são semelhantes aos *sockets* da família AF_PACKET, mas oferecem desempenho superior porque nenhuma cópia do pacote é enviada para a pilha de rede. Isso permite contornar parcialmente a pilha de rede do kernel (parcialmente porque ainda é necessário um programa XDP que redirecione o pacote, e a camada XDP faz parte da pilha de rede). Durante o restante deste trabalho, *sockets* deste tipo serão referenciados como *sockets XDP*.

Sockets XDP estão diretamente ligados a uma região de memória no espaço do usuário chamada UMEM (*User Mode Memory*). Essa região é dividida em *buffers* de tamanho igual, onde são armazenados os pacotes transferidos do/destinados ao kernel. O tamanho e a quantidade desses *buffers* podem ser configurados quando o programa de usuário aloca a memória para a UMEM. Além do conjunto de *buffers*, a UMEM também inclui dois *ring buffers*: o *fill ring* e o *completion ring*.

[3] https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/#driver-support
[4] https://netronome.com/agilio-smartnics/

O *fill ring* armazena os endereços relativos dos pacotes cuja posse foi transferida do espaço do usuário para o kernel, enquanto o *completion ring* armazena os endereços relativos dos pacotes que agora pertencem ao espaço do usuário. Aqui, endereço relativo denota o deslocamento do pacote a partir do *buffer* inicial do UMEM ao qual aquela posição no *ring buffer* se refere, conforme ilustrado na Figura 2.7.

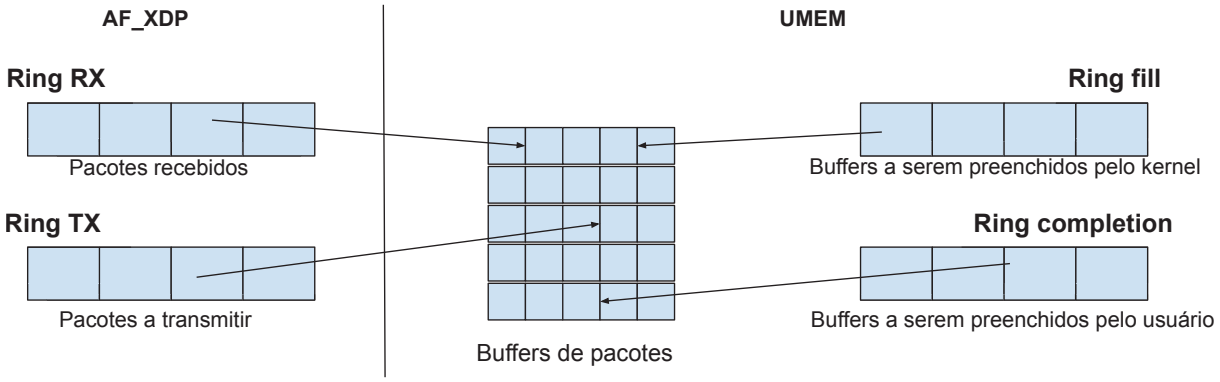


Figura 2.7: Arquitetura dos *sockets* XDP (Adaptado de Karlsson e Töpel (2018))

Um socket XDP, também alocado pelo usuário, possui dois *ring buffers*: RX e TX, para pacotes recebidos e transmitidos, respectivamente. Quando um pacote é recebido, o kernel preenche um descritor no *RX*. Esse descritor aponta para um *buffer* de pacote no UMEM onde o pacote está armazenado, indica o deslocamento dentro desse *buffer* até o início do pacote, e especifica seu tamanho.

Para receber pacotes por meio de sockets XDP, um programa de usuário deve primeiro escrever endereços relativos no *fill ring* e, em seguida, submetê-los ao kernel. O kernel então lê um endereço (o primeiro adicionado pela aplicação, seguindo a política FIFO) e grava os dados do pacote no endereço indicado na UMEM. Após isso, o kernel escreve esse endereço no *RX ring* do *socket* que recebeu o pacote. A aplicação (de usuário), então, lê o endereço do *RX ring* (que corresponde ao primeiro endereço que ela colocou no *fill ring*).

Os próximos passos são os seguintes:

- (i) A aplicação lê o pacote acessando o endereço especificado no UMEM.
- (ii) Após processar o pacote, a aplicação recicla o endereço, colocando-o de volta no *fill ring* para futuras recepções de pacotes.

Para verificar se há novos eventos no *socket*, mais especificamente, se o kernel escreveu alguma nova entrada no *rx ring*, o processo de usuário deve fazer repetidas chamadas à função *poll*, passando como parâmetro os descritores dos *sockets* associados às filas da placa de rede.

A mesma lógica se aplica ao envio de pacotes diretamente do espaço de usuário. Neste caso, o kernel preenche o *completion ring*, que será lido pelo processo de usuário. Este último então escreve os pacotes nos endereços UMEM apontados, e os disponibiliza no *TX ring* do *socket*. O *kernel* deve então transmitir os pacotes do *buffer* apontado, e escrever o endereço novamente no *completion ring*. Cada *socket* XDP é vinculado a uma fila na placa de rede. Ao inicializar um *socket*, o processo em espaço de usuário precisa especificar qual placa de rede e qual fila dessa placa estará associada ao *socket*. *Sockets* associados a filas diferentes não têm conhecimento do tráfego redirecionado pelos demais.

Cada *socket* é conectado a uma UMEM, mas uma única UMEM pode ser associada a múltiplos *sockets*, desde que todos estejam na mesma fila da placa de rede. Em outras palavras, a

UMEM está indiretamente associada a uma única fila na placa de rede. Todos os quatro tipos de *rings* descritos são estruturas de dados FIFO (*First-In-First-Out*). Os *rings fill* e *TX* são considerados produtores (escritos pela aplicação e lidos pelo kernel), enquanto os *rings RX* e *completion* são consumidores (escritos pelo kernel e lidos pela aplicação).

Para os produtores (*fill* e *TX*), a aplicação deve reservar espaços no *ring*. Uma vez que esses espaços estejam preenchidos com endereços relativos da UMEM, eles devem ser submetidos ao kernel. Para os consumidores, o programa do usuário precisa consultar o *ring*: se houver pacotes disponíveis, eles podem ser consumidos. Depois que os pacotes de um *slot* forem lidos, o *slot* deve ser liberado para que o kernel possa preenchê-lo novamente, permitindo assim uma reciclagem de endereços.

2.5 ALGORITMO AHO-CORASICK

O algoritmo Aho-Corasick é utilizado para busca de padrões em textos longos Aho e Corasick (1975). Esse algoritmo é eficiente para encontrar um número arbitrário de *substrings* em uma dada *string*. Portanto, se alinha bem aos objetivos de um IDS, haja vista que esses sistemas visam, entre outros objetivos, identificar assinaturas de *malware* no *payload* de um pacote. O Aho-Corasick é capaz de realizar a busca por padrões maliciosos em tempo linear com relação ao tamanho do *payload*. Por essa razão, o algoritmo é utilizado por Snort e Suricata Waleed et al. (2022).

A saída do algoritmo Aho-Corasick é um autômato capaz de reconhecer qualquer um dos padrões fornecidos como entrada em uma *string* qualquer. O autômato é construído de forma a criar enlaces de falha entre os estados: quando, no processamento da *string* de entrada, for encontrado um estado que não tem correspondência com o próximo caractere avaliado, o algoritmo procura por esse caractere num estado mais genérico, mas sem reiniciar a busca a partir da raiz do autômato. Mais especificamente, o enlace de falha de um nó *n* aponta para o maior sufixo próprio de *n* que também é prefixo de algum padrão. A construção deste tipo de enlace será detalhada e exemplificada na Subseção 2.5.2.

Com a intenção de explicar o funcionamento do algoritmo, a lista de padrões a serem adicionados no autômato, para posterior reconhecimento, será aqui chamada de *dicionário*. É possível definir as ações realizadas pelo Aho-Corasick em 3 principais passos:

- Criação de uma *trie*;
- Adição de *fail links*, na *trie* criada no passo anterior;
- Adição de *go links*, na *trie* criada no primeiro passo;

Cada uma dessas ações será detalhada nas seções a seguir.

2.5.1 Criação da *trie*

O Aho-Corasick armazena os padrões do dicionário em uma estrutura de dados *trie*, estrutura de dados em forma de árvore usada para armazenar e recuperar strings de um dicionário. Por exemplo, a adição dos padrões *abba, bb, bar, ar, foobar, foo* resultaria na árvore mostrada pela Figura 2.8. Os nós em verde representam estados finais.

2.5.2 Adição de *fail links*

Note que é possível aproveitar os sufixos na busca de padrões. Por exemplo, imagine que a *string* onde devem ser encontrados os padrões é *abar*. Neste caso, ocorreriam as transações

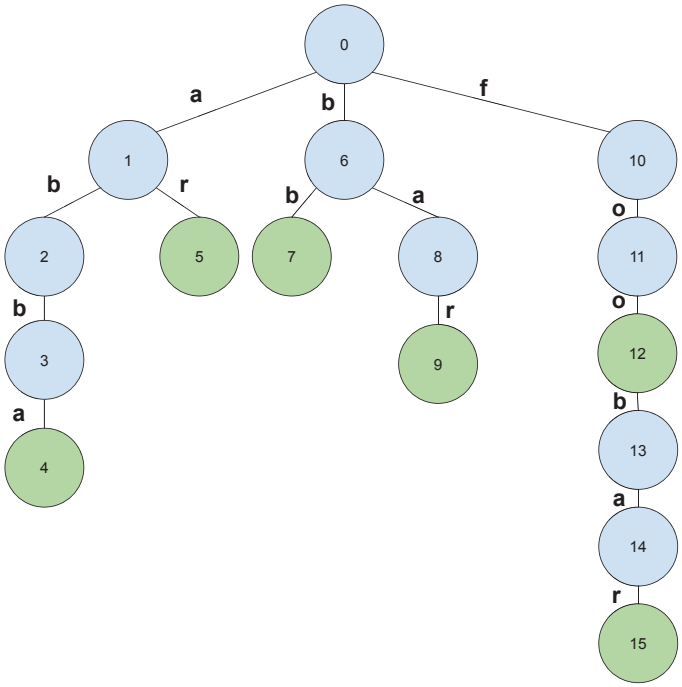


Figura 2.8: *Trie* para os padrões *abba, bb, bar, ar, foobar, foo*.

até o estado 2, onde não seriam mais encontradas possíveis transações, e o padrão *bar* não seria encontrado.

Para resolver este problema, são criados os *fail links*. Essa técnica cria um enlace do nó que representa o maior sufixo próprio de um padrão até o nó que representa este sufixo como o prefixo de outro padrão. Um sufixo próprio é definido como qualquer sufixo de uma *string* que tenha tamanho menor que o tamanho da *string*. Por exemplo, *cola* tem como conjunto de sufixos próprios *{ola, la, a}*. No caso apresentado como exemplo, é preciso ter um *fail link* do nó 2 para o nó 6. O nó 2 possui um sufixo próprio: $\{b\}$ (obtido recursivamente). O maior sufixo que pode ser considerado um prefixo de outro padrão é b. O algoritmo então procura pelo maior prefixo que possua b, que é representado pelo estado 6.

A mesma lógica pode ser aplicada para a construção do *fail link* do estado 3. Este estado possui dois sufixos próprios: $\{b, bb\}$. O algoritmo então procura pelo prefixo bb em outros estados. Ele encontra uma correspondência no nó 7, criando, portanto, um *fail link* do nó 3 para o nó 7. Se for considerado o nó 7, o sufixo b deve ser ligado ao prefixo b, representado pelo nó 6. A Figura 2.9 representa os *fail links* para o autômato avaliado. Durante a busca de padrões (já com o autômato completo), caso o algoritmo verifique que não há transações possíveis, ou seja, uma falha, ele passa para o estado indicado por este tipo de enlace.

Considere como exemplo a *string* de entrada *abar*. O algoritmo iria até o estado 2, consumindo o caractere a. Ao verificar que não existe enlace do estado 2 com o caractere a (terceiro caractere da *string*), o algoritmo acessa o *fail link*, passando para o estado 6. Note que, como a busca no caminho atual falhou, o algoritmo ainda não consumiu o caractere b. Ademais, já no estado 6, o algoritmo consome os demais caracteres da *string*, começando pelo b, até encontrar o estado final 9.

2.5.3 Adição de *go links*

Note que o algoritmo precisa retornar exatamente quantos padrões foram encontrados, e que existem alguns casos em que mais de um padrão estão presentes na *string* avaliada. No

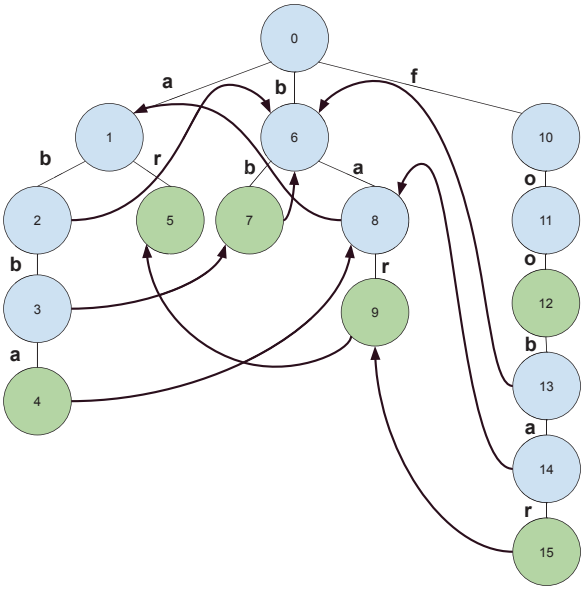


Figura 2.9: *Trie* com *fail links*.

autômato dado como exemplo, para a *string* de entrada *foobar*, os padrões *foo, foobar, bar, ar* devem ser encontrados.

Para isso, são adicionados os *go links*. Este tipo de enlace é responsável por apontar para estados finais. Por exemplo, no estado 3, o padrão *bb* já foi lido, portanto deve haver um *go link* do estado 3 para o estado 7. O mesmo se aplica ao estado 15, no qual o padrão *bar* já foi lido, e para o estado 9, no qual o padrão *ar* já foi lido.

A Figura 2.10 representa o autômato Aho-Corasick completo, com os *fail links* em vermelho e os *go links* em verde.

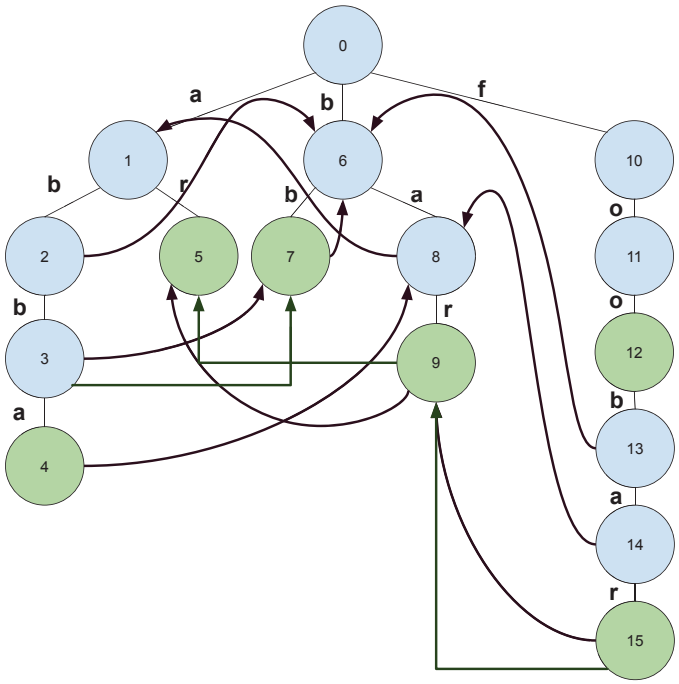


Figura 2.10: Autômato Aho-Corasick completo.

O algoritmo possui complexidade de tempo $O(n+m+z)$, onde n é o tamanho da *string* na qual os padrões estão sendo buscados, m é a soma dos tamanhos de todos os padrões, e z é o número de padrões encontrados nessa *string*. Portanto, pode-se concluir que, quanto menor o tamanho do autômato, melhor será o desempenho desse algoritmo.

2.6 SOFTIRQ

Para se comunicar com o sistema operacional da máquina *host*, dispositivos geram interrupções. Ao receber uma requisição de interrupção - IRQ (*Interrupt Request*), os circuitos do processador suspendem seu fluxo de execução corrente e desviam para um endereço predefinido, onde se encontra uma rotina de tratamento de interrupção (*interrupt handler*). Essa rotina é responsável por tratar a interrupção, ou seja, executar as ações necessárias para atender o dispositivo que a gerou. Ao final da rotina de tratamento da interrupção, o processador retoma o código que estava executando quando recebeu a requisição.

Por exemplo, ao receber um pacote de rede, o tratamento de interrupção que veio do controlador *Ethernet* (placa de rede) é elencado a seguir:

- o processador está executando um programa qualquer
- um pacote é recebido pela placa *Ethernet*
- o controlador *Ethernet* envia uma IRQ ao processador
- o processamento é desviado para a rotina de tratamento de interrupção
- a rotina de tratamento é executada para interagir com o controlador de rede (via barramentos de dados e de endereços) para transferir os dados do pacote de rede do controlador para a memória - DMA
- a rotina de tratamento da interrupção é finalizada. O processador retoma a execução do programa que foi interrompido.

Interrupções não são eventos raros, haja vista que ocorrem a cada recebimento de pacote, clique de mouse, operação de disco, entre outros. Visto isso, as rotinas de tratamento de interrupções devem ser curtas e realizar suas tarefas rapidamente; caso contrário, o desempenho do sistema pode ser prejudicado.

A listagem acima mostra somente o tratamento da interrupção até a transferência do pacote para a região de DMA em memória. Feito isso, deve existir uma rotina que trate do processamento da pilha de rede do kernel Linux (Figura 2.6). Como o tratamento da pilha não é uma tarefa que pode ser realizada rapidamente, haja vista que cada uma das camadas pode conter diversas rotinas de execução, ele não pode ser realizado por tratadores de interrupções comuns.

Para contornar este problema, este processamento é realizado por *softirqs*, que são tarefas de software executadas imediatamente após uma interrupção de hardware.

Toda a pilha de rede do kernel opera no contexto de *softirqs*. Para pacotes recebidos, o *softirq* responsável pelo processamento é *NET_RX_SOFTIRQ*. Como os programas XDP são inseridos na pilha de rede, eles também são executados no contexto de *softirqs*. Portanto, para medir o tempo de CPU consumido por um programa XDP, deve-se observar o tempo que o processador passa em contexto de *softirq*.

2.7 PERF EVENTS

Não existem muitas maneiras de enviar dados de um programa BPF, que está em espaço de kernel, para um processo de usuário. Uma delas dá-se através de *perf events*. Com o uso dessa abordagem, um usuário pode definir uma estrutura que será preenchida em espaço de kernel e lida em espaço de usuário. Para realizar a comunicação, deve ser criado um mapa BPF do tipo *BPF_MAP_TYPE_PERF_EVENT_ARRAY*.

É possível definir uma estrutura de dados que representa um pacote de rede, permitindo que um programa BPF seja capaz de enviar dados do pacote para espaço de usuário. Para enviar estes dados, o processo de usuário cria um *buffer* de *perf events*, para receber os eventos enviados pelo kernel. Para verificar se há novos eventos, o espaço de usuário executa repetidas funções *poll*, checando se há novos eventos no *buffer*.

3 REVISÃO DA LITERATURA

Desde o surgimento do Snort, em 1998, foram propostos diversos trabalhos que têm como principal objetivo tratar de sistemas de detecção de intrusão por assinatura. Alguns deles somente compararam duas ou mais soluções, buscando entender como características do tráfego, quantidade de regras carregadas, largura de banda e uso de CPU e memória afetam quanto do tráfego será de fato avaliado pelas soluções. Já outros tentam propor novos sistemas, comparando-os com outras soluções e mostrando de que maneira é possível superá-las. Existem também trabalhos que têm por objetivo propor técnicas capazes de aumentar o desempenho de alguma solução já existente.

Além de trabalhos relacionados diretamente a IDS, também é possível encontrar trabalhos relacionados às tecnologias utilizadas no presente manuscrito, entre elas destacam-se o XDP, os sockets da família XDP, e o BPF usado no contexto de processamento de tráfego.

Os trabalhos relacionados podem ser então separados em 3 conjuntos: (i) trabalhos que realizam comparação entre soluções IDS ou que tentam incorporar técnicas que, de alguma maneira, aumentem o desempenho de uma solução já existente; (ii) trabalhos que usam BPF e XDP para processamento de tráfego; e (iii) trabalhos que propõem uma nova arquitetura de sistema. O final deste Capítulo ainda conta com o detalhamento da arquitetura de aquisição e processamento de pacotes apresentada em Wang e Chang (2022), que é a base para a presente dissertação.

3.1 TRABALHOS QUE COMPARAM OU APRESENTAM TÉCNICAS PARA MELHORIA DE DESEMPENHO DE SOLUÇÕES IDS EXISTENTES

A maioria dos trabalhos desta categoria comparam os três sistemas de código aberto mais utilizados: Snort, Suricata e Bro (atualmente chamado de Zeek)[1]. O Snort 3, que é a versão *multithreaded* da ferramenta, foi lançada somente em 2021. Portanto, grande parte dos trabalhos compara o Snort *single threaded* com o Suricata, que é, desde sua gênese, um sistema *multi thread*. Enquanto Snort e Suricata são NIDS por assinatura, Zeek se define como um *Network Security Monitor*, capaz de realizar detecção de intrusões no tráfego de rede tanto por assinatura quanto por anomalia, tendo uma arquitetura *multi thread*.

Shah e Issac (2018) comparam o Snort *single threaded* com o Suricata *multi threaded*. Os resultados apontam que o Suricata é capaz de processar mais pacotes que o Snort em todas as larguras de banda avaliadas (de 1 a 10 Gbps), graças ao uso mais adequado de *hardware*. Hu et al. (2017) mostra a diferença entre as soluções (Suricata, Snort e Bro) quando executadas com suas configurações padrão *versus* quando executadas com configurações específicas. Mais notadamente, a maneira que a captura de pacotes é realizada afeta diretamente o desempenho de um NIDS: usando-se a biblioteca *libpcap* os piores resultados foram obtidos, seguida de *sockets* da família *AF_PACKET*, e obtendo o melhor resultado, a biblioteca de captura *pf_ring*.

Hu et al. (2020) buscam mostrar a viabilidade de usar um IDS (baseado em assinatura) de código aberto em redes de alta velocidade, com taxas de transmissão até 100 Gbps. O manuscrito mostra que a partir de 60 Gbps o uso dessas soluções se torna inviável, devido ao alto número de pacotes não avaliados. Os autores também ressaltam que devido ao alto tempo gasto pelo processador em contexto de *softirq* (processamento da pilha de rede), não é viável que o IDS compartilhe recursos computacionais com os *hosts* que devem receber o tráfego.

[1] https://zeek.org/

Zhengbing et al. (2008) mostram a viabilidade de avaliar regras já existentes do Snort a fim de derivar novas regras para ataques correlatos. Os autores afirmam que tal ação já seria possível através do algoritmo *Signature Apriori*, porém o algoritmo proposto por eles é mais eficiente, haja vista que o que mais toma tempo para derivar as novas regras é a análise da base das assinaturas, o algoritmo proposto é capaz de operar em bases reduzidas.

Wang et al. (2006) desenvolveram um sistema capaz de analisar tráfego trocado entre atacante e vítima de *spyware*, a fim de determinar novas regras do Snort para serem incorporadas ao IDS. O *NetSpy* possui uma arquitetura de dois módulos. O primeiro, chamado de módulo de diferenciação, analisa as diferenças entre tráfego benigno e tráfego infectado com *spyware*, fazendo-o através da avaliação dos cabeçalhos dos pacotes (existem *blacklists* de portas e endereços). O segundo módulo, chamado de gerador de assinaturas, é capaz de identificar a porção do *payload* do pacote que de fato representa a assinatura do *spyware* através do algoritmo *Longest Common Subsequence (LCSeq)*, usado para substituir *strings* variantes da assinatura, como por exemplo um domínio.

Salah e Kahtani (2010) descrevem quais as diferenças de desempenho do Snort quando executado nos sistemas operacionais Linux e Windows. O trabalho mostra em detalhes como funciona o subsistema de recepção de pacotes dos Linux, conhecido como New API (NAPI) Salim et al. (2001), apontando que o processamento dos pacotes ocorre na execução da *sofirq* `NET_RX_SOFTIRQ`. O trabalho ainda aponta que o Snort se comporta na maior parte do tempo como um processo associado a uso extensivo de CPU, pela execução do seu motor de detecção. Como a execução de *softirqs* também é caracterizada pelo uso extensivo de CPU, é possível que ambos os processos (a *softirq* e/ou o Snort) não consigam tratar todos os pacotes que deveriam, o que resulta em perda de pacotes para o primeiro, e pacotes não avaliados para o segundo.

O trabalho ainda mostra como a alteração de alguns parâmetros de sistema afeta o desempenho do Snort. No Windows, alterar o parâmetro *Processor Scheduling* a fim de fornecer mais tempo de processamento (CPU) para processos de *kernel* ou usuário não afeta o desempenho da solução. Já no Linux, descobriu-se que o valor padrão do parâmetro *NAPI budget* não é adequado para o bom desempenho do Snort, haja vista que o tempo de processamento oferecido ao IDS é inferior a aquele oferecido para as *softirqs*. Como conclusão geral, os autores afirmam que o Linux com o parâmetro modificado é superior ao Windows, principalmente se houver pacotes maliciosos em meio ao tráfego analisado. Contudo, para Windows e Linux padrão, uma instância do Snort executando no primeiro apresenta desempenho relativamente superior, quando não há tráfego malicioso.

Lin e Lee (2013) fazem uma análise minuciosa, evidenciando que o Snort é dividido em diversos pré-processadores que realizam tarefas diferentes. Os autores citam o *stream5*, responsável por remontar fragmentos TCP, o *frag3*, responsável por remontar segmentos IP, e o *http_inspect*, que normaliza os campos do cabeçalho HTTP. Para todos os tipos de tráfego avaliados, o tempo gasto no pré-processador *stream5* foi o maior entre todos os demais pré-processadores. Existe também um outro estágio, conhecido como *mpse*, que verifica qual subconjunto das regras carregadas pelo Snort deve ser avaliado para o pacote corrente. Tal verificação é feita ao analisar as opções das regras e os cabeçalhos do pacote corrente. Tem-se que, para tráfego HTTP não malicioso, 29.46% do tempo é gasto com pré-processamento, ao passo que 19.16% é usado para *mpse*, e 15.92% é destinado para percorrer a árvore de regras em busca de algum padrão malicioso. Os autores reforçam que estes tempos dependem do tipo de tráfego avaliado, mostrando que, para conexões SSH, o tempo para o estágio de *mpse* é baixo (0.1%), haja vista que o Snort, assim como outras soluções, não é capaz de encontrar padrões de *malware* em tráfego cifrado, encerrando assim o processamento do pacote neste estágio.

Ozkan-Okay et al. (2021) apresentam uma revisão sobre o estado da arte em detecção de intrusão. O trabalho referencia diversas técnicas e ferramentas usadas na detecção, para sistemas baseados em anomalias e assinaturas, e também para soluções de *host* e de rede. O artigo mostra tabelas elencando os principais trabalhos de cada uma das categorias de IDS descritas acima, como também disponibiliza uma pequena descrição de cada trabalho, mostrando brevemente qual foi a proposta e quais os resultados atingidos.

3.2 TRABALHOS QUE PROPÕEM PROCESSAMENTO DE TRÁFEGO COM BPF/XDP

Høiland-Jørgensen et al. (2018) apresentam o XDP, mostrando que, apesar do uso de sistemas de *driver* de espaço de usuário, como o DPDK (*Data Plane Development Kit*) [2], *Netmap* Rizzo (2012) e PF_RING [3], ser de fato um fator considerável para o ganho de desempenho, o contorno do kernel do sistema operacional traz problemas. Em mais detalhes, as verificações impostas pelo sistema também são contornadas, contornando também as medidas de proteção nativas oferecidas pelo Linux. O XDP é então introduzido como um *framework* para processamento rápido e seguro de pacotes, haja vista que programas XDP são submetidos ao verificador BPF. O trabalho mostra também a alta vazão proporcionada pelo *framework*, que é capaz de processar 24 milhões de pacotes por segundo, num computador com 6 núcleos de CPU.

Baidya et al. (2018) descrevem um sistema BPF capaz de realizar DPI num contexto de *stream* de vídeos e Internet das Coisas. O trabalho propõe uma arquitetura para filtragem de pacotes em um núcleo de CPU, e posterior escalonamento para processamento em outro, definindo assim uma política de balanceamento de carga. Os pacotes de interesse são aqueles que representam *frames* específicos. Caso esse seja o caso, o pacote é clonado e enviado para armazenamento. Os autores apontam que o desempenho do sistema é escalável, ou seja, mesmo que existam várias instâncias de processos esperando para realizar o processamento dos pacotes, o sistema BPF, que é uma tecnologia intrinsecamente paralela, é capaz de permitir que essas instâncias não fiquem ociosas. Contudo, o sistema funciona somente para pacotes específicos, com UDP como protocolo de transporte e MTU de menos de 200 bytes, diferente dos comumente encontrados na internet, o que não se adapta aos propósitos genéricos de um NIDS.

Xhonneux et al. (2018) demonstram que é possível executar funções de roteamento de pacotes de rede definidas por usuário usando BPF, permitindo assim que o processo seja realizado por uma máquina Linux comum. Os resultados mostram que a execução de programas BPF em roteadores Linux têm pouca ou nenhuma influência na quantidade de tráfego processado, não afetando o desempenho do sistema significativamente.

Scholz et al. (2018) mostram em detalhes a comparação de indicadores de custo e desempenho entre programas XDP e alternativas tradicionais de *firewall*, como *iptables* e *nftables*. Os resultados mostraram que, mesmo com as limitações do BPF/XDP da época, as soluções baseadas em XDP obtiveram desempenho superior quando comparados perda de pacotes e uso de CPU. Os autores também demonstram que mesmo com a superioridade do XDP para as abordagens tradicionais de *firewall*, o XDP não é capaz de superar alternativas que deixam de realizar o processamento da pilha de rede do *kernel*, como o DPDK. Porém, não deixam de citar que o DPDK, apesar de garantir baixa latência, mantém o uso de CPU em 100% devido às constantes operações de *pool* executadas pelo *driver* da tecnologia, independentemente da quantidade de pacotes que devem ser processados.

O trabalho desenvolvido por D et al. (2022) mostra o uso de BPF/XDP para cálculo de NAT, mais especificamente o mapeamento de IPv4 para IPv6 e vice-versa. Quando um pacote sai

[2] https://www.dpdk.org/

[3] https://www.ntop.org/products/packet-capture/pf_ring/

de um cliente (que usa IPv4) e tem como destino um servidor (com IPv6 habilitado), o roteador fará a tradução dos endereços do pacote, nesse caso, o processo de tradução chama-se NAT64. Os autores evidenciam que é possível realizar a tradução tanto do NAT64 quanto do NAT44 (IPv4 privado para público) usando o XDP, o que garante baixo *overhead* no processo de tradução e segurança para a execução, proporcionados pelo BPF.

Karlsson e Töpel (2018) evidenciam as diferenças no processo de redirecionamento de pacotes para espaço de usuário com *sockets* XDP e *drivers* de espaço de usuário, como o DPDK. A principal contribuição é referente à presença de um programa XDP, que é responsável pelo redirecionamento e avaliação do mapa BPF que guarda descritores para os *sockets*. O trabalho propõe uma maneira de não depender mais deste programa, que por executar na pilha de rede, usa recursos de processamento em kernel e *softirq*. Os resultados mostraram que, apesar da proposta ainda não superar o DPDK, supera a abordagem normal usada para o redirecionamento.

3.3 TRABALHOS QUE PROPÕEM NOVAS SOLUÇÕES DE IDS

Kostopoulos (2024) apresenta um NIPS BPF baseado em anomalias, no qual características comportamentais de ataques conhecidos foram extraídas e utilizadas no treinamento de um modelo de aprendizado de máquina. As estruturas que representam o modelo treinado são organizadas em mapas BPF e usadas para processar o tráfego de entrada para a detecção de intrusão. Usando o *payload* do pacote ingressante, bem como os pesos e viés do modelo, é possível calcular um valor chamado de '*Malicious Indicator Number*', que determina o veredito para aquele pacote. A solução foi comparada para quatro tipos de ataques, em termos de uso de CPU, com o Snort atuando como IPS, onde os pacotes considerados maliciosos seriam descartados. Enquanto o Snort usava 18% de CPU para ataques do tipo *OS Fingerprint*, a solução proposta usava pouco mais de 1%, onde ambas as soluções conseguiam detectar a íntegra dos pacotes maliciosos.

Apesar de apresentar bons resultados, este trabalho não deixa claro as condições de execução do Snort, por exemplo: (i) com quantas regras o sistema estava carregado; (ii) qual foi a largura de banda usada nesses testes; (iii) qual a abordagem de aquisição de pacotes foi configurada para obter estes resultados; (iv) qual a abordagem usada pelo autor para comparar a solução desenvolvida (um IPS baseado em anomalias/modelos de *machine learning*) com o Snort (um IDS baseado em assinaturas de *malware*/casamento de padrões). Visto isso, a presente dissertação busca ter o maior nível de clareza possível com relação aos detalhes das comparações e do ambiente de execução dos testes.

Uddin et al. (2013) mostram que, como as soluções por assinatura são dependentes da base, e que cada entrada da base deve ser avaliada para cada pacote processado, é interessante ter um conjunto menor e mais significativo de assinaturas. A ideia principal do sistema proposto é compartilhar as assinaturas que representam as ameaças mais comuns em determinado período com o detector de padrões. O detector ficaria então com uma pequena base de assinaturas, que deve ser atualizada periodicamente, obtendo assim um desempenho superior.

Baykara e Daş (2019) propõem um sistema por assinatura vinculado a um *honeypot*, chamado de *SoftSwitch*. O sistema foi desenvolvido utilizando *honeypots* de baixa, média e alta interação. O *SoftSwitch* tem como objetivo usar os *honeypots* para atrair potenciais intrusões. A solução tenta então detectar tentativas de intrusão específicas através de assinaturas previamente carregadas. Caso não encontre, um módulo do sistema fica responsável por analisar a intrusão, criando uma assinatura capaz de detectá-la. A abordagem levou a uma queda considerável na taxa de falsos positivos, o que beneficiou o monitoramento de uma rede corporativa em tempo real.

Malek et al. (2020) usa conceitos de sistemas especialistas (soluções baseadas em anomalias e aprendizado de máquina) para criar assinaturas. A ideia central do trabalho é propor um sistema que seja capaz de aprender o comportamento de um *malware*, e automaticamente criar uma base de assinaturas relacionada. Para isso, assim como as soluções de anomalia, é considerada uma base de comportamento normais. A partir disso, as detecções são avaliadas para a verificação de falsos-positivos. Se não houver, ou seja, se for realmente um *malware*, o sistema analisa suas características e cria uma assinatura.

Implementação de soluções IDS são dividas entre anomalia e assinatura. Otoum e Nayak (2021) propõem um sistema chamado de *AS-IDS*, ou *Anomaly and Signature IDS*, que combina as duas abordagens para a detecção de padrões de ataques conhecidos ou desconhecidos. O sistema é divido entre um subsistema que atua como IDS de assinatura, buscando por padrões sabidamente maliciosos, e um subsistema que atua como IDS de anomalia, usando algortimos de *machine learning* para a detecção de comportamentos maliciosos desconhecidos.

Wang e Chang (2022) introduzem uma solução NIDS BPF baseada em assinaturas (aqui denominada PF IDS), composta por dois programas: um executando no espaço de usuário e outro no espaço de *kernel* via XDP, para a detecção de *fast patterns*. Assim, apenas pacotes cujo *payload* contém um *fast pattern* são transferidos do espaço de *kernel* para o de usuário.

O desempenho do PF IDS será comparado com o do Sapo-boi com relação à quantidade de pacotes avaliados, uso de CPU e capacidade de comunicação entre espaços de kernel e usuário. Por isso, a arquitetura do sistema desenvolvido por Wang e Chang (2022) é detalhada a seguir.

3.4 ARQUITETURA DO PF IDS

O sistema proposto em Wang e Chang (2022), aqui chamado de PF IDS, é composto por um processo de usuário e um programa XDP. O processo de usuário carrega o programa BPF e popula os mapas, enquanto o programa XDP processa os pacotes ingressantes em busca de *fast patterns*.

A Figura 3.1 mostra a arquitetura do programa XDP do PF IDS, buscando evidenciar como o sistema encontra o mapa que deve ser avaliado para a detecção do *fast pattern*.

Nota-se que existem dois níveis de mapas. O mapa do primeiro nível recebe como entrada a porta fonte (TCP/UDP) encontrada ao verificar os cabeçalhos. Esse é um mapa do tipo vetor de mapas, ou seja, o retorno da requisição será um mapa BPF, classificado pelos autores como um mapa de nível 2.

O mapa de nível 2 recebe como entrada a porta destino encontrada anteriormente, retornando um inteiro, que representa um índice para acesso ao mapa *AC states*. Este último, por sua vez, é composto por mapas que representam o autômato Aho-Corasick (representados na figura como *SM - State Machines*). Em outras palavras, o resultado de um acesso ao mapa *AC states* é um mapa autômato, que representa o autômato de *fast patterns* que deve ser avaliado para o pacote corrente. Note que este mapa foi preenchido anteriormente, em espaço de usuário, antes da execução do programa XDP.

Para os protocolos ICMP e IP, os autômatos sempre serão os mesmos. Ou seja, o índice que os representa no mapa *AC states* é o mesmo, havendo um único autômato para cada um deles. Uma vez encontrado o mapa autômato, o PF IDS analisa o *payload* do pacote em busca de algum padrão, percorrendo todos os *bytes* do pacote. Ao encontrar padrões maliciosos, o sistema envia um *perf event* ao programa de espaço de usuário. Este evento contém, entre outras informações, uma cópia dos dados do pacote.

Quando o espaço de usuário recebe o pacote (realizando operações de *polling*, numa estrutura de memória compartilhada, para onde são enviados os eventos), termina de verificá-lo,

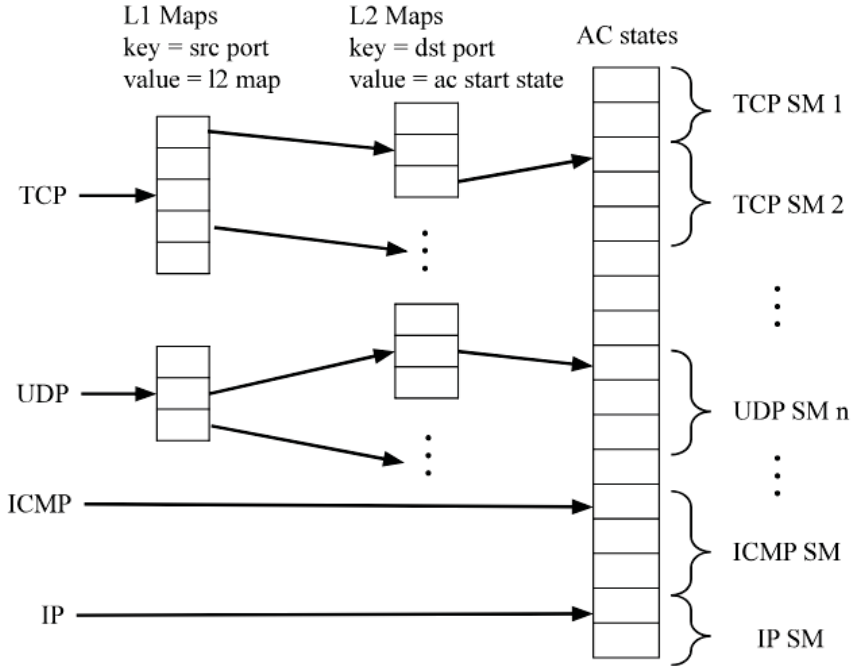


Figura 3.1: Arquitetura do PF IDS. Disponível em Wang e Chang (2022).

a fim de encontrar os padrões faltantes para gerar o alerta com base na regra que contém o *fast pattern* encontrado.

O PF IDS, ao contrário do Sapo-boi, não suspende a execução do programa XDP ao encontrar o primeiro *fast pattern*. Para todos os pacotes, o sistema percorre todos os *bytes* do *payload* do pacote corrente. Se mais *fast patterns* forem encontrados, o sistema envia um *perf event* para cada coincidência, mas desta vez sem a cópia dos dados do pacote.

Essa abordagem aumenta a tendência da solução deixar de avaliar pacotes em espaço de usuário. Como é discutido no Capítulo 6, os *perf events* são estruturas genéricas para o envio de informação de espaço de kernel para usuário, que possuem elevadas taxas de perda, ou seja, parte dos eventos enviados não é recebida. Como o PF IDS pode mandar eventos em demasia, a probabilidade de que os eventos que contenham dados do pacote sejam perdidos aumenta.

Devido à indisponibilidade de código-fonte, algumas especificidades da implementação do PF IDS foram inferidas pelo autor desta dissertação. A principal delas é que não está claro no trabalho como os autores identificam, em espaço de usuário, qual regra deve ser avaliada, ou seja, qual *fast pattern* foi encontrado em espaço de kernel. Para contornar este problema, os eventos enviados pelo espaço de kernel contêm, além dos dados do pacote, dados para a recuperação da regra.

Os resultados de Wang e Chang (2022) mostram uma comparação entre o PF IDS e o Snort. Evidencia-se que o PF IDS possui maior vazão que o Snort, ou seja, para uma mesma quantidade de regras carregadas, o Snort passa a deixar de avaliar pacotes a uma largura de banda menor que o PF IDS. Os autores também mostram que o PF IDS faz uso menor de CPU em contextos de *softirq*, *kernel* e usuário do que o Snort.

Soluções clássicas, como Snort e Suricata possuem uma arquitetura de aquisição e processamento de pacotes assim como descrita na Figura 3.2

Em espaço de kernel são definidas as rotinas de aquisição de pacotes. O Snort o faz por meio da biblioteca *libpcap*, ao passo que o Suricata realiza esta tarefa através dos *sockets AF_PACKET*. Ambas as abordagens enviam cópias dos pacotes ao espaço de usuário. Contudo,

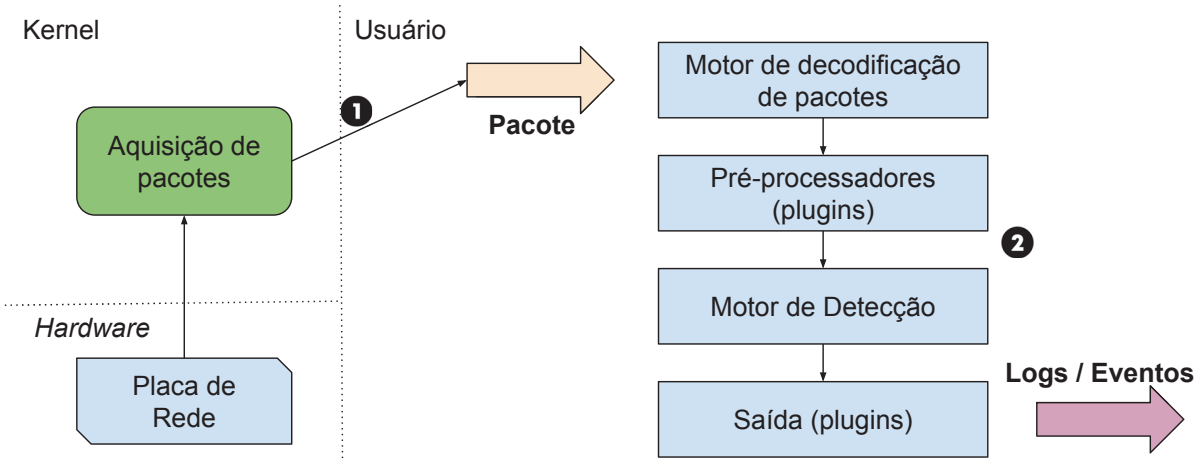


Figura 3.2: Arquitetura das Soluções Clássicas.

o Snort possui uma única instância da *libpcap* para o envio dos pacotes recebidos por todas as CPUs, ao passo que o Suricata usa um *socket AF_PACKET* por CPU, garantindo um processo de aquisição de pacotes realmente paralelo.

Já o primeiro passo em espaço de usuário é a execução do *motor de decodificação de pacotes*, responsável por armazenar as informações dos cabeçalhos em estruturas externas. Em seguida, tem-se os *pré-processadores*, que funcionam como *plugins*. Três deles merecem destaque: o *frag3*, capaz de remontar pacotes fragmentados; o *session*, responsável por manter controles sobre os fluxos TCP e UDP; e o *stream5*, capaz de realizar remontagem de fluxo.

O *Motor de Detecção* é quem de fato realiza a busca de assinaturas maliciosas. Assim como discutido na Seção 2.1.3, os *fast patterns* devem ser os primeiros padrões a serem buscados pelo sistema. Já o módulo de saída é responsável por gerar os alertas e escrevê-los nos arquivos de *log*, no terminal, ou então enviá-los a outro processo.

A grande contribuição de Wang e Chang (2022) é evidenciar a possibilidade (e não a viabilidade, devido às limitações dos experimentos discutidos no Capítulo 1) de se construir um sistema de detecção de intrusão em kernel, usando BPF/XDP, seguindo uma arquitetura diferente das soluções clássicas. Os autores propuseram a arquitetura mostrada pela Figura 3.3, que também é usada pelo Sapo-boi.

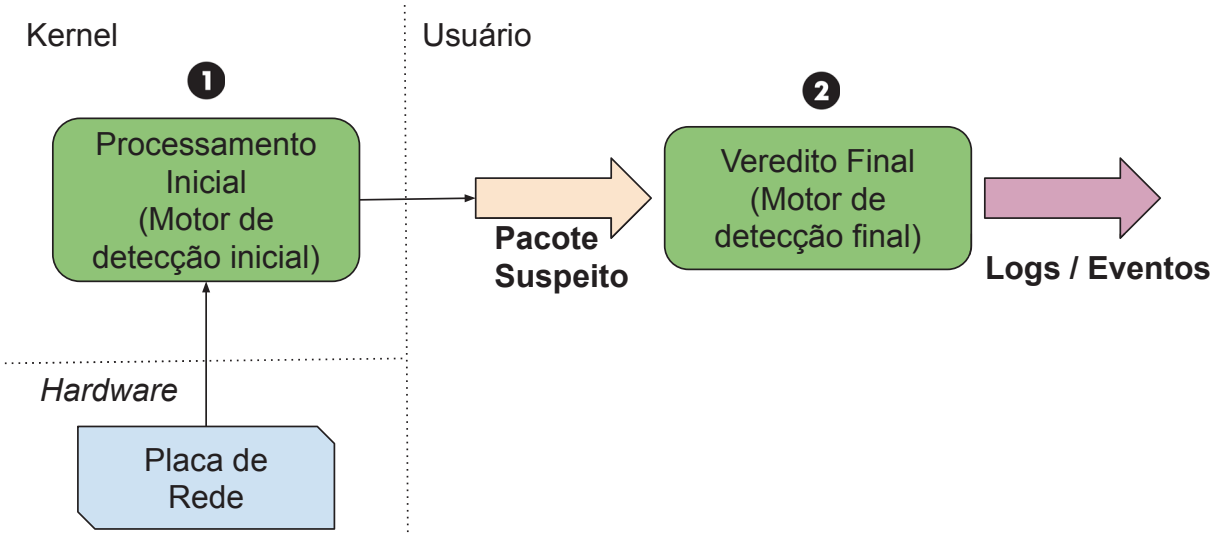


Figura 3.3: Arquitetura Proposta por Wang e Chang (2022).

A principal diferença é o deslocamento do Motor de Detecção para o espaço de usuário, permitindo a detecção dos *fast patterns* em espaço de kernel, o que viabiliza que somente os pacotes considerados suspeitos cheguem ao espaço de usuário. Essa abordagem permite que a não haja enfileiramento significativo de pacotes em espaço de usuário, que é a principal razão pelas quais as soluções clássicas (Snort e Suricata) perdem pacotes, assim como evidenciado pelos resultados presentes no Capítulo 6.

3.5 CONSIDERAÇÕES FINAIS

Após o levantamento dos trabalhos relacionados, fica claro o aumento de desempenho (sobre diferentes métricas) que o BPF e XDP podem proporcionar num contexto de redes de computadores. Os trabalhos puramente de comparação de soluções IDS mostram que o Suricata é um IDS robusto e de bom desempenho, superando seu principal concorrente, o Snort, em métricas como número de pacotes avaliados, uso de CPU e uso de memória.

Visto isso, tem-se que o Suricata representa a melhor opção de IDS por assinatura de código aberto disponível. Já para os trabalhos que propõem soluções de IDS novas, o que mais se destaca é o PF IDS Wang e Chang (2022). Tal sistema é capaz de superar o Snort em vazão e uso de CPU, mas não foi comparado com o Suricata. A proposta de um IDS por assinatura baseado em XDP/BPF, e a comparação deste em termos de número de pacotes avaliados, com Snort, Suricata e PF IDS ainda não haviam sido contempladas em uma única solução.

4 O DETECTOR DE INTRUSÕES SAPO-BOI

A solução proposta neste trabalho, denominada Sapo-boi, é um sistema de detecção de intrusão em redes por assinatura, constituído por dois módulos principais:

- Módulo de Suspeição, implementado como um programa XDP em execução em espaço de kernel/*driver*, responsável por identificar *fast patterns* a partir das regras carregadas;
- Módulo de Avaliação, um processo de usuário, cujo objetivo é determinar se outros padrões (*contents*) relacionados à regra referenciada pelo *fast pattern* detectado no Módulo de Suspeição estão presentes no pacote suspeito.

A Seção 4.1 busca detalhar a arquitetura de alto nível da solução desenvolvida no mestrado, detalhando brevemente os dois módulos, ao passo que mostra como ocorre a cominicação entre eles. A Seção 4.2 trata especificamente do Módulo de Suspeição, evidenciando como é realizada a busca de padrões em espaço de kernel. Finalmente, a Seção 4.3 detalha o Módulo de Avaliação, mostrando como é realizado o recebimento dos pacotes suspeitos, bem como o procedimento para o veredito final e a geração de alertas.

4.1 ARQUITETURA EM ALTO NÍVEL DO SAPO-BOI

A Figura 4.1 ilustra a arquitetura em alto nível do Sapo-boi. No lado esquerdo, está representada a fase de inicialização do sistema. Inicialmente, o Módulo de Avaliação é responsável por carregar o Módulo de Suspeição (programa XDP) no kernel, bem como os mapas BPF que esse componente utilizará. Visto isso, o Módulo de Avaliação deve processar as regras e preencher duas estruturas: (i) mapas BPF, que representam autômatos Aho-Corasick construídos a partir dos *fast patterns* das regras; e (ii) os autômatos de *contents*, estruturas de grafos que residem no espaço de usuário até o fim da execução do NIDS, responsáveis pela detecção de padrões nos pacotes considerados suspeitos.

Após o preenchimento de todos os mapas BPF e estruturas em espaço de usuário, passa-se para a fase de Execução (porção direita da figura), na qual os pacotes ingressantes são processados paralelamente no Módulo de Suspeição. Se o pacote for considerado suspeito, ele é redirecionado para espaço de usuário via *sockets XDP* para posterior processamento no Módulo de Avaliação. Se este último chegar ao veredito de que o pacote é de fato malicioso, escreverá no arquivo de logs, indicando qual foi a regra que gerou o alerta.

Cabe ressaltar que, como NIDS tradicionais, a solução proposta processa cópias dos pacotes; ou seja, na ocorrência de uma ação `XDP_DROP` (descarte de um pacote), o tráfego original não é afetado de nenhuma forma. Com isso, se nenhum *fast pattern* for encontrado pelo Módulo de Suspeição (i.e., o pacote não foi considerado suspeito), a cópia do pacote em análise é imediatamente descartada e não requer processamento adicional do Módulo de Avaliação. Por outro lado, se houver um *fast pattern* em um pacote sob análise, o Módulo de Suspeição o envia para o Módulo de Avaliação através de *sockets* XDP. Para isso, a solução baseia-se em mapas BPF do tipo XSKMAP, que devem ser criados e inicializados a partir do Módulo de Avaliação, permitindo a comunicação entre os dois módulos. Uma vez que os mapas BPF são preparados, o Módulo de Avaliação entra em modo de espera, aguardando a chegada de pacotes suspeitos. As seções a seguir detalham as operações realizadas por cada um dos módulos do Sapo-boi.

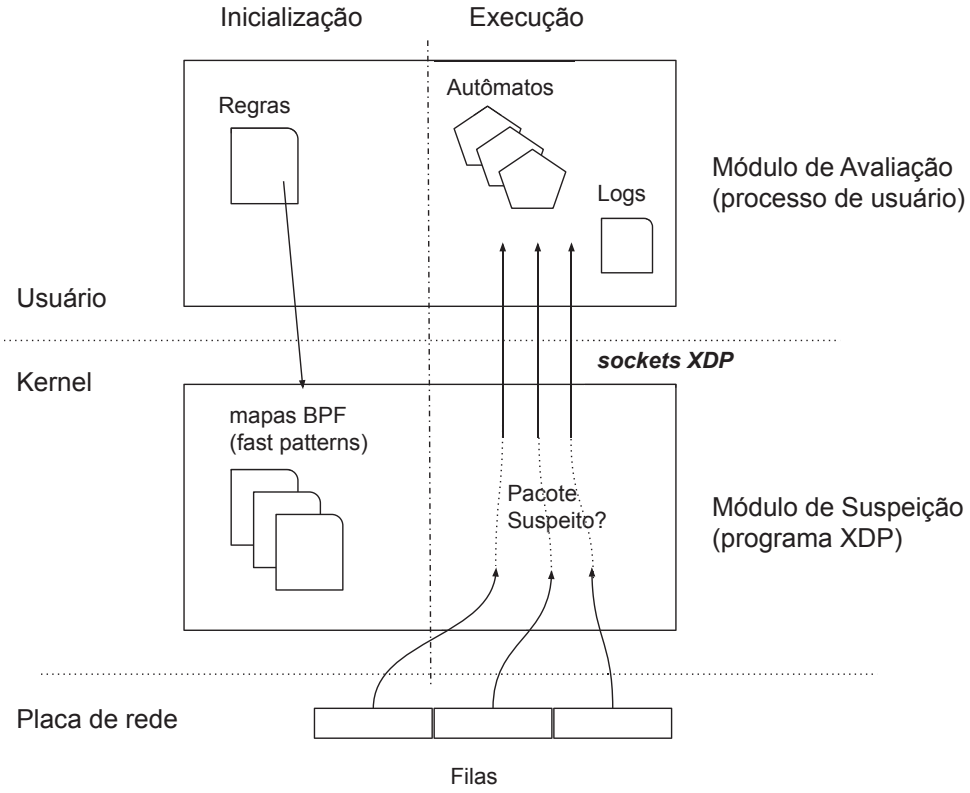


Figura 4.1: Arquitetura do Sapo-boi.

4.2 MÓDULO DE SUSPEIÇÃO

O Módulo de Suspeição recebe o tráfego de rede diretamente da interface, sem qualquer processamento prévio, ou seja, a leitura e interpretação dos cabeçalhos dos pacotes são feitas pelo próprio Módulo (por exemplo, ao considerar um fluxo TCP, ele deve identificar inicialmente o cabeçalho Ethernet, seguido pelos cabeçalhos IP e TCP, para finalmente ser capaz de inspecionar o *payload*).

Caso o tráfego de entrada apresente encapsulamento com *tags* VLAN, o Módulo de Suspeição descartará essas *tags* até a detecção do campo *ethertype* do cabeçalho Ethernet correspondente. No Linux, as VLANs são configuradas criando interfaces virtuais do tipo vlan; mas, como o programa XDP é executado diretamente na interface real, ele verá todos os pacotes com suas *tags* VLAN antes que o kernel atribua o pacote às interfaces virtuais.

Uma vez obtido o cabeçalho Ethernet, localiza-se a opção *ethertype* (próximo protocolo), referente à camada de rede. Se o protocolo observado não for IP, o pacote é descartado; senão, o módulo considerará dois casos possíveis para a camada de transporte: TCP e UDP. Outros possíveis protocolos de transporte resultam no descarte do pacote.

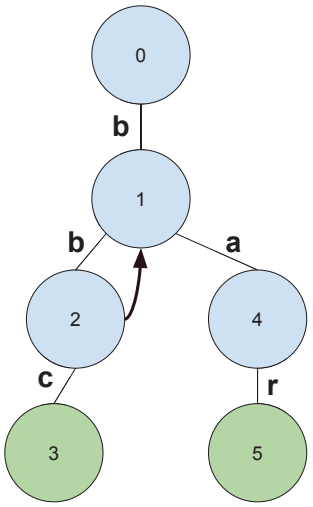
Para que o Módulo de Suspeição execute de maneira eficiente, múltiplos *cores* de CPU devem ser usados concomitantemente. Isso só é possível se a placa de rede tiver suporte a múltiplas filas (permitem o uso de *sockets* XDP) e *Receive Side Scaling* (RSS) Woo e Park (2012).

4.2.1 Detecção de Padrões em Espaço de Kernel

Terminada a identificação dos cabeçalhos do pacote recebido, o Módulo de Suspeição verifica a existência dos *fast patterns* por meio de mapas BPF do tipo *hash* (*BPF_MAP_TYPE_HASH*). Esses mapas permitem que sejam definidas estruturas do tipo

chave/valor, que representam estados de um autômato Aho-Corasick. Particularmente, as chaves representam um estado do autômato (inteiro) e uma transição (caractere/byte), enquanto os valores definem o estado resultante da transição, uma *flag* que indica se este novo estado é final ou não, e o *fail link*. Note que uma transição de estado no autômato Aho-Corasick é feita pelo Módulo de Suspeição ao verificar se uma chave pertence ao mapa. Como os mapas do tipo *hash* são otimizados para busca, a transição é feita em tempo constante. Note também que, caso o pacote avaliado tenha seu *payload* criptografado, o casamento de assinaturas se torna inviável no contexto desta solução.

A Figura 4.2 mostra a tradução de um exemplo de autômato Aho-Corasick para um mapa BPF do tipo *hash*. O autômato possui adicionados os padrões $\{bbc, bar\}$. Nota-se que os *fail links* também são representados, para simulação do comportamento do algoritmo através de mapas. A estratégia para popular os mapas (o que deve ser realizado em espaço de usuário) será discutida na Seção 4.3.



Chave		Valor		
Estado	**Transição**	**Novo Estado**	**É final?**	**Fail Link**
0	b	1	Não	0
1	b	2	Não	0
1	a	4	Não	0
2	c	3	Sim	1
4	r	5	Sim	0

Figura 4.2: Representação de um Autômato Aho-Corasick em uma mapa BPF.

4.2.2 Grupos de Portas UDP/TCP

O algoritmo Aho-Corasick se torna mais eficiente para autômatos menores, pois tanto o tamanho total dos padrões quanto o número potencial de combinações de padrões são reduzidos, conforme discutido na Seção 2.5. Note que mesmo que o algortimo de busca de padrões seja linear em relação ao tamanho do *payload* dos pacotes, o desempenho do algoritmo ainda é influenciado pela quantidade de padrões carregados, bem como pela quantidade de padrões presentes no *payload*.

Logo, para criar autômatos enxutos, usa-se a estratégia de segmentação por grupos de portas TCP/UDP, na qual cada grupo representa um par de portas origem e destino. As regras também são segmentadas por protocolo de transporte (UDP e TCP), compondo autômatos ainda menores. Cada par origem/destino é chamado de um grupo de portas.

Os grupos de portas fornecem uma forma de separar as regras para que o autômato gerado contenha apenas os *fast patterns* das regras referentes a um grupo específico. A Figura 4.3 exibe um exemplo da separação das regras em pares de portas. Os *fast patterns* do conjunto de regras que compõem um grupo de portas são mapeados para um único autômato.

Note que existem dois tipos de grupos de pares de portas: um para o UDP e outro para o TCP. É possível observar que, para o protocolo UDP, a regra com *signature id* (sid) 4 está

Figura 4.3: Agrupamento de regras por portas de origem e destino.

presente em todos os grupos de portas, isto é, em todos os autômatos, uma vez que não especifica suas portas de origem e destino de interesse. Para o TCP, o mesmo ocorre para a regra com sid 8. Esse fenômeno pode comprometer o desempenho do NIDS, pois o *fast pattern* correspondente à regra de sid 4, para UDP, e à regra de sid 8, para TCP, serão testados para todo o tráfego de entrada.

Todos os *fast patterns* de regras que estão no mesmo grupo são adicionados como padrões maliciosos no mesmo autômato, que por sua vez será transformado num mapa BPF, assim como mostrado pela Figura 4.2. Em outras palavras, o autômato do grupo i é criado com os *fast patterns* das regras que foram adicionadas ao grupo i.

Observe que cada grupo está relacionado a um par de portas. Por exemplo, a Figura 4.3 mostra que o grupo 0 do protocolo UDP está relacionado ao par de portas $(13, 42)$. Como cada grupo será transformado em um mapa BPF que representa o autômato, verifica-se que há uma mapeamento um pra um entre um par de portas e um mapa autômato.

Quando o Módulo de Suspeição verifica que um pacote usa TCP ou UDP como protocolo de transporte, ele obtém as portas de origem e destino do cabeçalho e analisa somente o autômato de interesse para este par de portas. Se nenhum autômato for encontrado, o pacote é descartado por não haver regra carregada que possa considerá-lo malicioso.

Os autômatos referentes aos grupos de portas definidos ficam disponíveis para o Módulo de Suspeição por meio de um mapa BPF, como mostra a Figura 4.4.

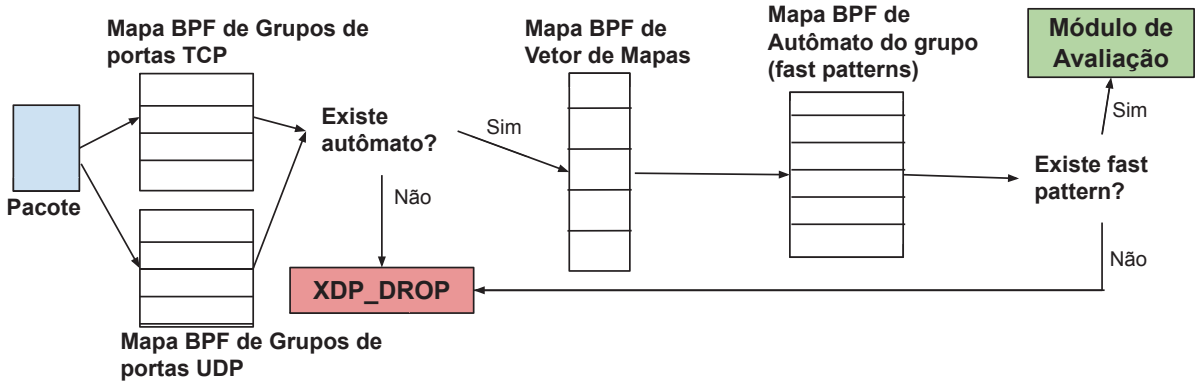


Figura 4.4: Caminho de um Pacote no Módulo de Suspeição.

Cada entrada do mapa referenciado na Figura 4.4 como *Mapa BPF de Grupos de Portas* possui como chave uma estrutura composta por dois inteiros de 16 bits, representando as portas de origem e de destino. O valor armazenado para esta chave é um índice para um mapa BPF de vetor de mapas. Cada posição do *Mapa BPF de vetor de Mapas* é um mapa autômato para um par de portas específico, como pode ser verificado na Figura 4.5. Existe um mapa de grupos de portas para UDP e outro para TCP

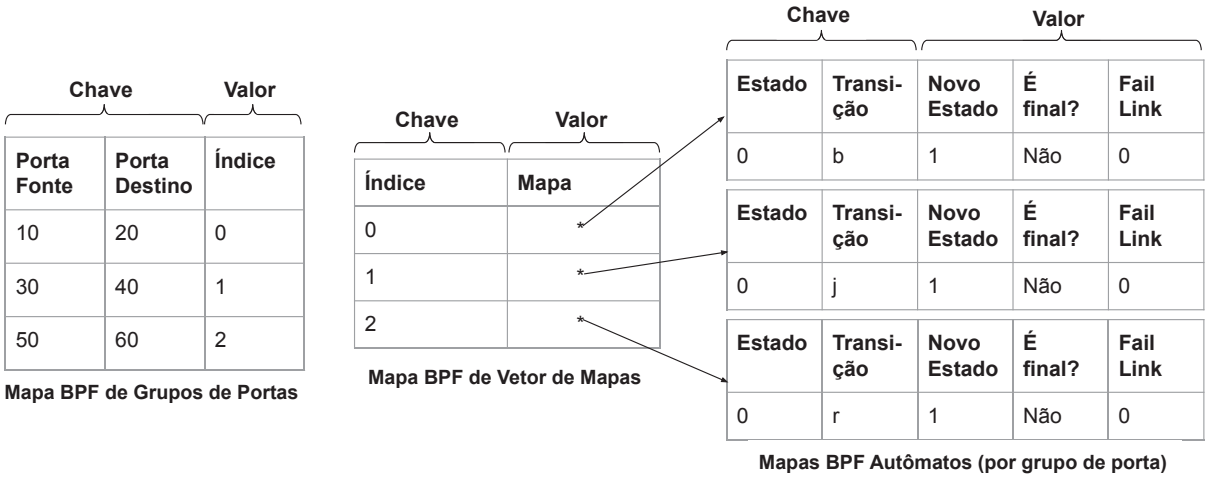


Figura 4.5: Especificação dos mapas do Módulo de Suspeição.

Portanto, quando um pacote é recebido, primeiramente verificam-se os protocolos até encontrar o par de portas TCP/UDP. Feito isso, o par de portas é verificado contra o mapa BPF de grupos de portas específico para o protocolo de transporte recebido. Se não existir entrada, o pacote deve ser descartado, por não haver regra carregada que contenha o mesmo par de portas que o pacote avaliado correntemente. Se existir autômato, o Módulo de Suspeição recebe um índice ao fazer a verificação do par de portas.

Este índice deve ser usado para acessar o *Mapa BPF de Vetor de Mapas*. Este tipo de mapa é estruturado de maneira que cada posição aponta para um mapa BPF. No caso da solução proposta, cada posição aponta para um mapa autômato, específico para um par de portas.

A partir do mapa autômato, o Módulo de Suspeição avalia o *payload* do pacote. Se um *fast pattern* for encontrado a partir do processamento do pacote em análise, ele é enviado ao Módulo de Avaliação; caso contrário, o pacote é descartado, assim como indicado na Figura 4.4.

4.2.3 Metadados

O Mapa autômato, além de conter as transições discutidas na Subseção 4.2.1, ainda contém um campo adicional: um índice para a regra cujo *fast pattern* foi encontrado. Esse índice é definido e preenchido em espaço de usuário, enquanto o mapa está sendo criado. Mais especificamente, quando o Módulo de Avaliação (em espaço de usuário) está populando os mapas, antes da execução do Módulo de Suspeição, as regras são agrupadas por par de portas. Em espaço de usuário, existe um vetor de pares de portas, e cada par de portas é definido como um vetor de regras.

Logo, quando está preenchendo os mapas, o Módulo de Avaliação adiciona um campo extra no mapa para os estados finais, indicando qual é o índice da regra que contém aquele *fast pattern*. Este índice é usado posteriormente para a recuperação rápida da regra que deve ser avaliada, quando o pacote é redirecionado para o espaço de usuário.

A Figura 4.6 mostra a organização das estruturas. Na porção de cima da figura, é possível perceber que o espaço de usuário constrói o autômato de um grupo de portas com base nos *fast patterns* das regras daquele grupo. Quando preenche o mapa, com base no autômato, o Módulo de Avaliação aproveita para adicionar o índice da regra cujo *fast pattern* é representado pelo estado final do autômato.

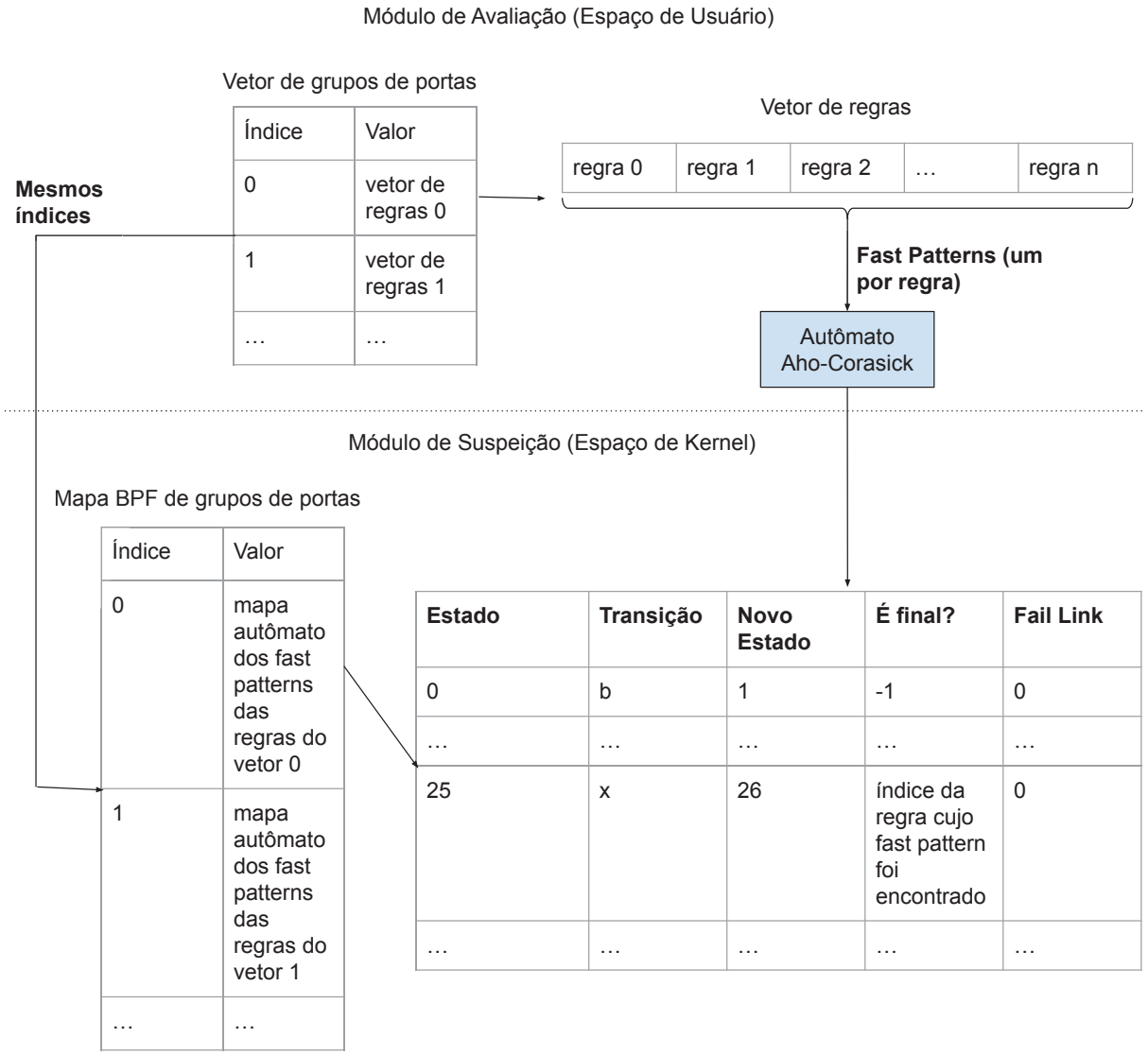


Figura 4.6: Correspondência de índices entre os módulos.

Também é possível perceber na figura que o índice do vetor de grupo de portas (de usuário) é o mesmo índice do mapa BPF de grupo de portas. Ou seja, o espaço de kernel tem acesso aos índices de ambas as estruturas: vetor de grupo de portas, que é obtido pelo mapa de grupos de portas, e índice do vetor de regras, obtido quando o programa encontra um *fast pattern*.

Ao encontrar um *fast pattern* em um pacote, o Módulo de Suspeição redireciona a íntegra deste, além de um conjunto de metadados, para o Módulo de Avaliação. No contexto XDP, os metadados são definidos por uma estrutura abstrata de dados adicionada antes do início do pacote. Para tanto, o programa redefine o ponteiro para o início do pacote e, em seguida, calcula o tamanho da estrutura, aumentando assim o espaço de dados conforme o espaço extra necessário. Essa operação é realizada através da função `bpf_xdp_adjust_meta`.

Como o envio dos pacotes se dá via *sockets* XDP, os metadados que os acompanham devem ser representados por estruturas compatíveis com BTF (*BPF Type Format*), o que implica considerar um alinhamento de 4 bytes para serem suportadas pelo *kernel*.

De maneira geral, o Módulo de Suspeição envia quatro informações como metadados:

- (i) o índice no vetor de mapas, obtido por meio do mapa de grupos de portas;
- (ii) o índice da regra identificada, determinado pelo *fast pattern* encontrado;
- (iii) uma sinalização que indica o protocolo da camada de transporte (TCP ou UDP).
- (iv) o deslocamento até o início do *payload* do pacote.

Essas informações permitem ao Módulo de Avaliação identificar eficientemente a regra que deve ser avaliada.

Assim que todas as etapas representadas na Figura 4.4 forem concluídas, ocorre o encaminhamento do pacote suspeito e de seus metadados para o Módulo de Avaliação por meio de *sockets* XDP. Para isso, é necessário existir um mapa do tipo *XSKS_MAP*, no qual uma chave corresponde ao identificador da fila da placa de rede na qual o pacote foi recebido e o respectivo valor designa o descritor do *socket* incumbido do envio. O processo de envio é realizado utilizando a função `bpf_redirect_map`.

4.2.4 Visão geral do Módulo de Suspeição

Como se pôde ver nas seções anteriores, o Módulo de Suspeição é um programa XDP capaz de buscar o par de portas de um pacote e um mapa, que representa um autômato Aho-Corasick associado ao par de portas encontrado. Este programa também é responsável por redirecionar pacotes para o Módulo de Avaliação caso um padrão malicioso seja encontrado.

O Algoritmo 1 mostra como a busca do par de portas e do mapa do autômato relacionado ocorre. Note que o programa XDP recebe um ponteiro para o início do pacote e atualiza a posição deste ponteiro ao passo que processa os cabeçalhos. O Módulo de Suspeição analisa cada cabeçalho até encontrar o par de portas TCP/UDP e, em seguida, usa essa informação para procurar o mapa de autômatos apropriado no vetor de mapas BPF discutidos na Seção 4.2.2. Além disso, o Algoritmo 1 recebe outras duas entradas: o vetor de mapas TCP, e o vetor de mapas UDP. Note que nas linhas 26 e 28, o algoritmo verifica esses mapas para encontrar o autômato específico para o par de portas e protocolo de transporte avaliado.

O algoritmo mostra ainda que há suporte para ambas as versões do protocolo IP, bem como que o tratamento para cada uma dessas versões deve ser diferente, devido às diferenças das estruturas que representam o cabeçalho de cada uma das versões. Observe que os pacotes em si não precisam ser modificados para servir como entrada para o algoritmo Aho-Corasick; é necessário apenas analisá-los até atingir a área de *payload*. Se nenhum mapa de autômato for encontrado no conjunto específico de mapas, o pacote é descartado, o que significa que o par de portas no pacote não possui nenhum autômato a ser analisado.

Por outro lado, o Algoritmo 2 descreve a busca no mapa autômato. Como entrada, tem-se o port_pair_index (índice do vetor de mapas), o deslocamento até atingir o *payload* do pacote, uma sinalização de qual é o protocolo de transporte (se não for TCP, é UDP) e o *automaton_map*, mapa específico que representa o autômato Aho-Corasick para o par de portas encontrado conforme descrito no Algoritmo 1.

Os dois primeiros parâmetros são necessários porque são enviados como metadados, conforme discutido na Seção 4.2.3. O terceiro parâmetro é o mapa autômato, onde devem ser encontrados padrões de *malware*.

Algoritmo 1 Ação do Módulo de Suspeição a cada pacote ingressante:

Require: *pkt_start*: ponteiro para o início do pacote
Require: *TCP_map_array*: Vetor de Mapas TCP
Require: *UDP_map_array*: Vetor de Mapas UDP

1: $eth\_type \leftarrow parse\_eth\_hdr(pkt\_start)$ // obter campo *ethertype* (IPv4 ou IPv6?)
2: $pkt\_start \leftarrow pkt\_start + sizeof(eth\_hdr)$ // deslocamento do ponteiro até o final do cabeçalho *Ethernet*
3: **if** $eth\_type = IPv4$ **then**
4: $ip\_type \leftarrow parse\_ipv4\_hdr(pkt\_start)$ // obter o campo *ip_type* (UDP ou TCP?)
5: $pkt\_start \leftarrow pkt\_start + sizeof(IPv4\_hdr)$ // deslocamento do ponteiro até o final do cabeçalho IPv4
6: **else if** $eth\_type = IPv6$ **then**
7: $ip\_type \leftarrow parse\_ipv6\_hdr(pkt\_start)$ // obter o campo *ip_type* (UDP ou TCP?)
8: $pkt\_start \leftarrow pkt\_start + sizeof(IPv6\_hdr)$ // deslocamento do ponteiro até o final do cabeçalho IPv6
9: **else**
10: **return** XDP_DROP // Caso pacote não contenha protocolo IP
11: **end if**
12: **if** $ip\_type = TCP$ **then**
13: $is\_TCP \leftarrow$ TRUE
14: $src\_port, dst\_port \leftarrow parse\_tcp\_hdr(pkt\_start)$ // obter as portas de origem e destino TCP
15: $pkt\_start \leftarrow pkt\_start + sizeof(TCP\_hdr)$ // deslocamento do ponteiro até o final do cabeçalho TCP (início do *payload*)
16: **else if** $ip\_type = UDP$ **then**
17: $is\_TCP \leftarrow$ FALSE
18: $src\_port, dst\_port \leftarrow parse\_udp\_hdr(pkt\_start)$ // obter as portas de origem e destino UDP
19: $pkt\_start \leftarrow pkt\_start + sizeof(UDP\_hdr)$ // deslocamento do ponteiro até o final do cabeçalho UDP (início do *payload*)
20: **else**
21: **return** XDP_DROP // Caso pacote não seja TCP nem UDP
22: **end if**
23: $ports.src\_port \leftarrow src\_port$
24: $ports.dst\_port \leftarrow dst\_port$
25: **if** is_TCP **then**
26: $automaton\_map, port\_pair\_index \leftarrow lookup(TCP\_map\_array, ports)$ // obter o autômato TCP de *fast patterns* para a porta encontrada
27: **else**
28: $automaton\_map, port\_pair\_index \leftarrow lookup(UDP\_map\_array, ports)$ // obter o autômato UDP de *fast patterns* para a porta encontrada
29: **end if**
30: **if** $automaton\_map = NULL$ **then**
31: **return** XDP_DROP // Caso nenhum autômato de *fast pattern* tenha sido encontrado
32: **end if**

A correspondência de padrões é feita iterando sobre cada *byte* do *payload* do pacote. Se houver uma correspondência entre o estado atual e o *byte* avaliado, o estado deve ser atualizado. Se um estado final for alcançado, o programa aloca espaço para metadados do pacote, preenche as informações descritas na Seção 4.2.3 (recebidas como entrada) e envia o pacote para o espaço do usuário por meio de *sockets* XDP. Observe que a função auxiliar *bpf_redirect_map* recebe um descritor para um *socket* XDP, obtido do identificador da fila na qual o pacote foi recebido. Também é importante destacar que se nenhum padrão for encontrado ao percorrer todo o *payload*, o pacote será descartado.

Algoritmo 2 Ação do Módulo de Suspeição ao Procurar por Padrões Maliciosos:

Require: *port_pair_index*: Índice do vetor de mapas, buscado pelo Algoritmo 1
Require: *offset*: Deslocamento no pacote até encontrar o *payload*
Require: *is_TCP*: Sinalização do protocolo de transporte, calculado pelo Algoritmo 1
Require: *automaton_map*: Mapa autômato específico do par de portas e protocolo de transporte, buscado pelo Algoritmo 1

```
1: xsk_desc ← XDP socket descriptor for this NIC queue
2: state ← 0
3: for byte ∈ packet_payload do
4:     state ← lookup(automaton_map, state, byte)
5:     if state.is_final then
6:         meta ← bpf_xdp_adjust_meta()
7:         meta.rule_index ← state.rule_index
8:         meta.port_pair_index ← port_pair_index
9:         meta.offset ← offset
10:        meta.is_tcp ← is_TCP
11:        return bpf_redirect_map(xsk_desc)
12:    end if
13: end for
14: return XDP_DROP
```

4.3 MÓDULO DE AVALIAÇÃO

O Módulo de Avaliação não se limita ao processamento de regras baseadas nas suspeitas levantadas pelo Módulo de Suspeição; ele executa uma série de operações antes mesmo que este último entre em funcionamento, mais notadamente a criação de mapas BPF, juntamente com o registro de metadados; o processamento das regras carregadas; as ações envolvendo *sockets* XDP; e o gerenciamento e processamento de pacotes suspeitos, redirecionadas pelo Módulo de Suspeição. Todas essas operações são detalhadas a seguir.

4.3.1 Criação dos Mapas e Registro de Metadados

A primeira operação realizada pelo Módulo de Avaliação é o carregamento do arquivo objeto BPF e a vinculação do programa XDP a uma interface de rede específica. Essa operação não apenas carrega o Módulo de Suspeição, mas constrói todos os mapas presentes no arquivo objeto (*i.e.*, cria os mapas descritos na Seção 4.2, porém ainda não os inicializa). Pode-se considerar que, num primeiro momento, o Módulo de Avaliação funciona como um carregador BPF, assim como mostrado na Figura 2.4.

A segunda operação refere-se ao registro das estruturas de metadados BTF. Tais estruturas possibilitam o envio de dados do espaço de memória do *kernel* para o espaço do

usuário, desde que todos os campos sejam devidamente registrados junto ao *kernel* e as restrições de alinhamento sejam cumpridas. Ou seja, as estruturas que representam os metadados, indicadas na Seção 4.2.3, devem ser registradas a posteriori. Para que essa operação seja realizada com sucesso, tais estruturas devem cumprir uma restrição de alinhamento em 4 *bytes*.

4.3.2 Processamento das Regras

Após a criação dos mapas e o registro de metadados, ocorre o processamento das regras. As operações executadas com base nas regras carregadas determinam os mapas de grupos de portas, vetor de mapas e autômatos de grupo (Figuras 4.4 e 4.5). O primeiro passo é determinar os grupos de portas (Figura 4.3), estruturas que contêm, entre outros elementos, um vetor de regras. Os grupos em si também são armazenados em um vetor específico, que diferencia o tráfego pelo protocolo da camada de transporte, ou seja, existe um vetor de grupos de portas para o TCP e um para o UDP.

Cada regra de um grupo de portas é tratada de forma distinta em relação ao *fast pattern* e aos outros *contents*. O *fast pattern* de cada regra é armazenado em uma lista, que posteriormente será convertida em um autômato e, em seguida, em um mapa BPF de um grupo de portas específico. A Figura 4.7 mostra os *fast patterns* do grupo de portas *i* sendo inseridos no mesmo mapa autômato. Além das informações de estado e transições, este mapa BPF terá, em suas entradas que representam um estado final do autômato, um campo indicando qual regra foi correspondida, ou seja, o índice da regra no vetor de regras do grupo, assim como mostrado pela Figura 4.6.

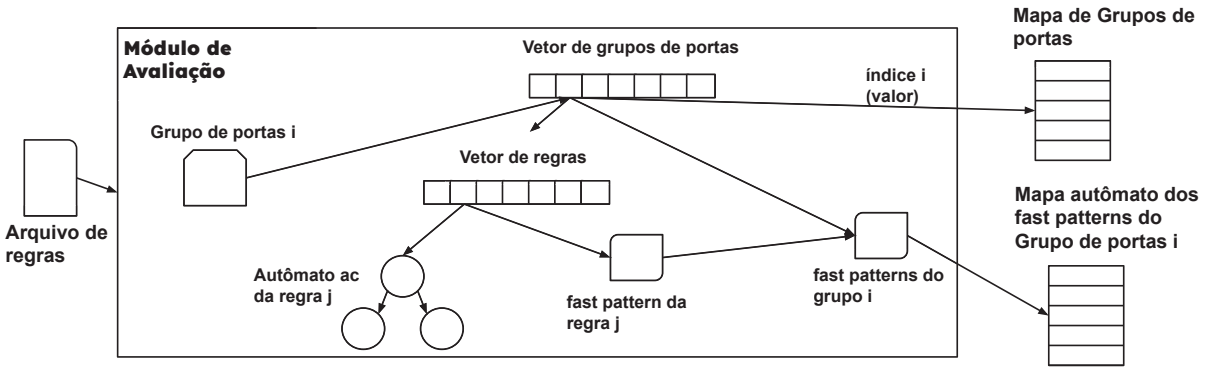


Figura 4.7: Estruturas do Módulo de Avaliação.

Logo, quando o Módulo de Suspeição encontra um *fast pattern*, ele é capaz de enviar ao Módulo de Avaliação, como metadados de um pacote, o índice do vetor do grupo de portas, além do índice do vetor de regras daquele grupo que indica a regra que deve ser avaliada. Portanto, o Módulo de Avaliação consegue encontrar a regra desejada em tempo constante.

Desta maneira, próximo passo é adicionar as portas e o índice do grupo no mapa BPF de grupos de portas (note que esse é o mapa que possui como chave um par de portas e como valor o índice no vetor de mapas, vide Figura 4.5).

Dessa forma, um mesmo índice:

- no vetor de mapas em *kernel*, representa o mapa que contém os *fast patterns* para aquele grupo de regras;
- no vetor de grupos de portas em espaço de usuário, representa o grupo de portas cuja regra tem como *fast pattern* o mesmo que foi encontrado pelo programa XDP.

Os *contents*, por outro lado, são tratados individualmente: para cada uma das regras do grupo, um autômato Aho-Corasick será criado (diferentemente dos *fast patterns*, para os quais um autômato é criado por grupo de portas). É importante observar que o autômato individual das regras não deve mais ser representado por mapas, uma vez que será analisado no espaço do usuário, passando a ser representado por estruturas de grafos.

4.3.3 Operações AF_XDP

Após a preparação das regras e padrões, as estruturas de *socket* e UMEM (região de memória compartilhada entre os módulos que conterá os pacotes suspeitos) são iniciadas, assim como o mapa de *sockets* XDP.

Para isso, inicialmente, é alocada memória para a região da UMEM onde estarão os *buffers* de pacotes. Nesta proposta, foram alocados 4096 *frames* de tamanho 4096 bytes, resultando em uma área de 16MB por UMEM. Este tamanho foi escolhido para garantir que, mesmo com altas taxas de transmissão e envio pelo Módulo de Suspeição, a UMEM ainda possua espaço para receber novos pacotes. Em seguida, a UMEM é registrada no *kernel* através da função `xsk_umem__create`. Note que cada *socket* possui uma UMEM associada.

Portanto, o próximo passo é o criar o *socket* XDP associado a UMEM alocada. Essa operação é realizada com o auxílio da função `xsk_socket__create`. Após a criação do *socket* e sua vinculação com uma UMEM, é necessário adicionar endereços relativos (deslocamentos em um *buffer*, conforme apresentado na Seção 2.4) no *ring fill* da UMEM. Esses endereços indicam ao *kernel* onde escrever os pacotes suspeitos.

O Sapo-boi permite que o usuário defina quantas e quais são as filas da placa de rede deseja utilizar para criar os *sockets*. Com as UMEMs e *sockets* iniciados, finalmente estes são adicionados ao mapa de *sockets* XDP. Para isso, é adicionado um índice inteiro positivo como chave no mapa, sendo este número representativo do índice da fila da placa de rede onde o *socket* será executado. O valor relacionado à chave é o descritor de arquivo do *socket*, obtido através da função `xsk_socket__fd`.

4.3.4 Gerenciamento e Processamento de Pacotes

Para receber pacotes, o Módulo de Avaliação executa a função *poll()*, que recebe como argumento uma lista de descritores de *socket* e retorna o número de descritores onde há um evento, neste caso, o evento é uma escrita no *RX ring* do *socket*, indicando que há pacotes disponíveis.

Em caso positivo, o Módulo de Avaliação, com o deslocamento escrito no *ring* RX e auxílio da função `xsk_ring_cons__rx_desc`, obtém o endereço relativo, assim como o tamanho, do pacote recebido. Então, o espaço de memória coletado do *ring* RX é devolvido ao *ring fill* da UMEM como disponível para reuso.

Com as informações de endereço relativo e tamanho, a função `xsk_umem__get_data` é utilizada para obter o endereço absoluto e, consequentemente, o pacote e os metadados enviados pelo Módulo de Suspeição. Finalmente, os pacotes e metadados são processados.

A Figura 4.8 mostra os metadados enviados para o espaço de usuário, e como eles são interpretados. Na porção esquerda da figura, é possível verificar os metadados recebidos pelo espaço de usuário.

Ao receber um pacote, o Módulo de Avaliação primeiro verifica qual o protocolo de transporte. Visto isso, ele acessa o vetor de grupo de portas daquele protocolo, na posição indicada pelo metadado. Ao acessar o grupo de portas específico, ele acessa a regra específica, por meio de um acesso no vetor de regras na posição indicada pelo índice do vetor de regras, que

também veio nos metadados do pacote. Por fim, o sistema usa a informação do deslocamento até o *payload* para saber por onde começar a verificar pelos padrões.

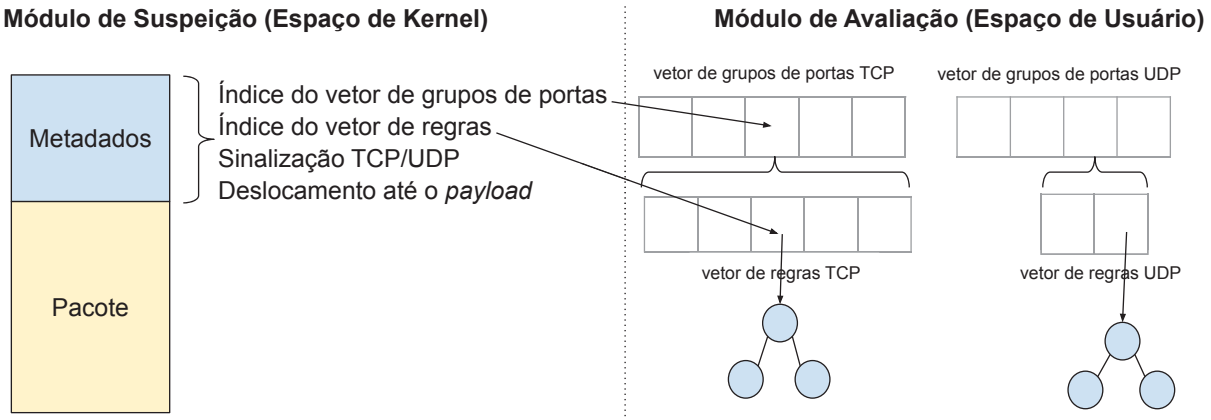


Figura 4.8: Interpretação dos Metadados.

Uma vez identificada a regra, o Módulo de Avaliação tem acesso ao autômato que contém os padrões daquela regra, e ao deslocamento ao qual deve realizar a busca pelos padrões restantes. Caso todos os *contents* sejam encontrados ao processar o *payload* do pacote, um alerta é gerado.

A seguir, os passos que o Módulo de Avaliação segue ao receber um pacote são resumidos:

- Os metadados são processados e interpretados. Isso significa que a aplicação sabe qual é o procolo de transporte; qual é o par de portas de interesse; qual autômato deve ser analisado; e qual o deslocamento dentro do pacote que representa o início do *payload*.
- Se o Módulo de Avaliação determinar que a regra não possui *contents* adicionais além do *fast pattern* previamente detectado, a regra é aceita, um alerta é gerado e o processamento é encerrado.
- Se houver padrões a serem analisados, o Módulo de Avaliação converte o *payload* do pacote em um vetor de caracteres e o analisa utilizando o autômato associado ao *fast pattern* previamente detectado da regra.
- Se o número total de padrões encontrados for igual ao número de padrões registrados na regra em análise, a regra é aceita, um alerta é gerado e o processamento é encerrado. Caso contrário, o pacote é considerado falsamente suspeito, nenhum alerta é gerado neste momento, e o Módulo de Avaliação avalia novamente o *payload* do pacote recebido, em busca de possíveis *fast patterns* diferentes daquele identificado pelo Módulo de Suspeição. Se houver novos *fast patterns*, os *contents* restantes da regra correspondente a cada padrão detectado serão avaliados. Caso contrário, o processamento é encerrado.

5 AMBIENTE DOS EXPERIMENTOS

Os experimentos têm como foco a comparação de quatro soluções NIDS: (i) Sapo-boi, desenvolvida como parte deste trabalho; (ii) PF IDS, descrita em Wang e Chang (2022), mas reimplementada neste trabalho devido à indisponibilidade do código-fonte; (iii) Snort, versão 3.81.84; e (iv) Suricata, versão 7.0.5.

O PF IDS é dividido em um programa XDP e um processo em espaço de usuário. O sistema permite a detecção de *fast patterns* em espaço de kernel, descartando os pacotes considerados seguros.

É importante destacar que a solução proposta em Wang e Chang (2022) é aqui chamada de "PF IDS" porque ela utiliza eventos ***perf** events* para transferir informações de pacotes do espaço de kernel para o espaço de usuário, enquanto o Sapo-boi realiza o redirecionamento de pacotes por meio de sockets XDP. A arquitetura do PF IDS está detalhada no Capítulo 3.

As comparações são baseadas principalmente em três fatores:

- (i) a taxa de pacotes não analisados;
- (ii) o uso de CPU nos contextos de kernel, usuário e *softirq*;
- (iii) o número de pacotes suspeitos perdidos durante a comunicação entre os módulos (espaços de kernel e usuário), para o Sapo-boi e o PF IDS.

Adicionalmente, este capítulo descreve a topologia de rede utilizada nos testes, juntamente com as especificações de seus componentes. É importante observar que, durante os experimentos, cada NIDS foi executado individualmente sob o mesmo conjunto de regras, utilizando o mesmo hardware e o mesmo software subjacente. Em outras palavras, todos os esforços foram feitos para garantir as mesmas condições de execução para as quatro soluções.

Com relação às soluções baseadas em XDP no espaço de kernel (Sapo-boi e PF IDS), os testes foram conduzidos com a adição de uma operação de cópia de memória no *driver*, imediatamente antes da execução do programa BPF. Esse ajuste foi necessário para emular o processo de aquisição de pacotes realizado pelas soluções em espaço de usuário, que operam sobre tráfego copiado. Em outras palavras, o kernel utilizado nos testes do Sapo-boi e do PF IDS difere ligeiramente do kernel padrão usado nos testes com o Snort e o Suricata. A principal diferença é a inserção de uma chamada à função `memcpy` no *driver mlx4*, antes da invocação do programa XDP.

As Seções a seguir apresentam detalhes sobre o ambiente experimental de testes e os softwares auxiliares utilizados nos experimentos, bem como a arquitetura do programa XDP do PF IDS.

5.1 HARDWARE E SOFTWARE

O ambiente no qual os experimentos foram executados é descrito da seguinte forma: um *host* executa o NIDS avaliado, enquanto um gerador de tráfego envia pacotes para o mesmo em diferentes taxas de transmissão (largura de banda). A Tabela 5.1 apresenta as especificações dos dois componentes mencionados (ambos com especificações idênticas). As placas de rede das duas máquinas estão conectadas por meio de um cabo bidirecional *Small Form-factor Pluggable (SFP+)*.

Tabela 5.1: Descrição dos *hosts* usados nos experimentos.

Sistema Operacional	*Ubuntu 24.04 LTS; Kernel 6.8.0-31-generic; x86_64*
Placa de Rede	*Mellanox Technologies MT27500 [ConnectX-3] 10Gbps*
CPU	*12th Gen Intel(R) Core(TM) i7-12700; 16 cores*
RAM	16GB
Enlace	SPF+ E124936-D cobre bidirecional

5.2 REGRAS E CONFIGURAÇÃO

As regras utilizadas nos experimentos correspondem a um subconjunto modificado do *Snort Registered Ruleset* [1]. O Snort apresenta 3 classes de conjuntos de regras: *Community Ruleset*, regras gratuitas para *download*, sem necessidade de registro; *Registered Ruleset*, constituído de regras gratuitas, mas há a necessidade de registro no site do Snort; e por fim, regras *Subscription*, na qual os usuários devem pagar pelo conjunto de regras, com a vantagem de tê-las atualizadas todos os dias. O conjunto *Registered Ruleset* foi escolhido devido ao fato de ser atualizado mais frequentemente, além de possuir um conjunto maior de regras com acesso disponível de forma gratuita.

Conforme descrito em Wang e Chang (2022), determinados tipos de regras foram removidos para formar o subconjunto utilizado nos experimentos. Primeiramente, foram excluídas as regras que não possuíam a opção *content*, haja vista que o Sapo-boi e o PF IDS precisam de ao menos um *content*, que seria o *fast pattern*, para conseguir carregar a regra.

Em seguida, também foram removidas as regras que exigiam *plugins* (pré-processadores) e/ou módulos específicos do Snort. Mais especificamente, o Snort é composto por quatro módulos principais, que são executados após a fase de aquisição, para cada um dos pacotes.

Além disso, o Snort permite regras com *contents* e portas negados; tais regras também foram excluídas. Ou seja, existem regras que aceitam como *content* qualquer *string*, exceto aquela especificada. O mesmo vale para os pares de portas.

O subconjunto resultante foi utilizado para testar o Sapo-boi, o PF IDS e o próprio Snort. No caso do Suricata, foram necessárias pequenas modificações nessas regras, uma vez que, apesar de similar, Snort e Suricata não compartilham a mesma sintaxe para a construção das regras.

Com relação às configurações, todos os pré-processadores, *plugins* e módulos do Snort e do Suricata que não estavam diretamente envolvidos na detecção de padrões foram desativados, exceto aqueles responsáveis por exibir estatísticas de processamento ao final da execução do NIDS. O Snort e o Suricata foram testados utilizando seus modelos padrão de captura de pacotes. Especificamente, o Snort foi testado com a biblioteca *libdaq*, que fornece uma camada de abstração sobre a *libpcap*, enquanto o Suricata realizou a captura de pacotes utilizando *sockets AF_PACKET*.

5.3 GERAÇÃO E ENVIO DE TRÁFEGO

Para gerar o tráfego dos experimentos, foi feito um programa que analisa as regras e forja pacotes maliciosos utilizando a biblioteca Python Scapy. Além disso, o utilitário *iperf3* foi empregado para gerar tráfego não malicioso. O tráfego gerado por essas ferramentas foi armazenado em arquivos `pcap`.

[1] https://www.snort.org/downloads/#rules

Apesar do uso de tráfego real, como o disponibilizado em Pei et al. (2025), representar mais fielmente o que pode ser encontrado em redes usuais, os pacotes que o compõem não poderiam ser relacionados aos padrões de assinaturas e portas endereçados pelo subconjunto de regras usados nos experimentos.

Como um dos objetivos deste trabalho é comparar o desempenho computacional da solução proposta com as demais, optou-se por ter total controle sobre o tráfego usado nos testes, a fim de ser capaz de verificar como a quantidade de tráfego malicioso afeta cada uma das soluções.

Ademais, como é exibido no Capítulo 6, o tráfego malicioso corresponde a 5% do tráfego total gerado para testar as soluções. Um relatório feito pela *Cloudflare* aponta que 5% de todo o tráfego avaliado pelas soluções da empresa foi mitigado nos Estados Unidos em 2024 Belson (2025). Isso indica que, de todo o tráfego avaliado durante um ano por todos os sistemas da empresa, não só os por assinatura, 5% representavam atividade maliciosa.

Contudo, sabe-se que as soluções por assinatura somente são capazes de classificar como malicioso as assinaturas presentes na base. Como as assinaturas podem ser bastante específicas, uma solução dificilmente acusará como malicioso uma quantidade significativa de pacotes, quando comparada com a quantidade total de tráfego analisado White et al. (2013).

Por isso, a taxa de pacotes maliciosos escolhida para realizar os experimentos pode ser justificada, de maneira a representar um índice alto de indicativos de *malware*, quando consideradas as soluções por assinatura.

O tráfego é enviado do gerador para o *host* que executa o NIDS avaliado por meio da ferramenta `tcpreplay`, que permite configurar a largura de banda do enlace. Isso possibilita submeter a solução NIDS avaliada a diferentes taxas de recepção ao longo de intervalos de tempo, permitindo sua análise sob níveis variados de carga.

6 RESULTADOS

Este Capítulo apresenta e discute os resultados obtidos a partir dos experimentos realizados considerando o ambiente experimental descrito no Capítulo 5. Para isso, os experimentos descritos no capítulo anterior foram empregados, a fim de comparar a solução proposta com os demais NIDS.

A Seção 6.1 busca evidenciar as diferenças entre a capacidade de processamento de pacotes das quatro soluções. A primeira abordagem é analisar como o número de fluxos no tráfego afeta a quantidade de pacotes que a solução pode avaliar. Verifica-se que, devido às tecnologias presentes nas placas de redes modernas usadas para balanceamento de carga, pacotes pertencentes ao mesmo fluxo (uma sessão TCP ou uma conversa UDP) são processados pelo mesmo núcleo de CPU quando chegam à pilha de rede do kernel. Isso pode de fato prejudicar o desempenho de um IDS, haja vista que existirão núcleos de CPU que estarão sobrecarregados, enquanto os demais permanecerão ociosos, afetando assim a capacidade de aquisição de pacotes da solução.

O segundo ponto a ser avaliado é a quantidade de regras carregadas. Como é esperado, quanto maior a quantidade de assinaturas na base, mais processamento o IDS precisa realizar para encontrar padrões maliciosos em um único pacote. Contudo, para baixas larguras de banda, as soluções tendem a apresentar um bom desempenho, ao passo que para taxas de recebimento maiores, mais pacotes deixam de ser avaliados.

A Seção 6.2 busca evidenciar as diferenças de uso de CPU das soluções. Optou-se por coletar métricas, a diversas larguras de banda, para o uso de CPU em três diferentes contextos: usuário, kernel e *softirq*. Evidencia-se que existe uma discrepância significativa entre os valores das soluções em kernel e usuário, explicada principalmente pela arquitetura das soluções em kernel, na qual os pacotes não suspeitos são imediatamente descartados.

Por fim, a Seção 6.3 compara a solução proposta neste trabalho com o PF IDS, outro sistema que usa o XDP. Nota-se que, pela maneira que o Sapo-boi e o PF IDS redirecionam pacotes suspeitos ser diferente, existem diferenças entre a quantidade de pacotes considerados suspeitos em kernel que de fato chegam até o espaço de usuário para o veredito final. Mostra-se que a abordagem com *sockets* XDP, usada pelo Sapo-boi, supera a abordagem usada pelo PF IDS, baseada em *perf events*.

6.1 TAXA DE PACOTES NÃO ANALISADOS

Esta Seção detalha duas abordagens utilizadas para verificar a taxa de pacotes não analisados pelos sistemas. Pacotes não analisados são definidos como aqueles que chegam ao *host* destino, onde a solução NIDS está localizada, mas que, devido à altas taxas de transmissão, à quantidade de regras ou à natureza do tráfego, a solução não consegue capturar a íntegra destes pacotes para avaliação. Para simplificar, a taxa de pacotes não analisados será referida como a taxa de perda de pacotes da solução.

A primeira abordagem refere-se ao número de fluxos presentes no tráfego transmitido. Neste contexto, o número de fluxos determina quantos pares diferentes de endereços IP e porta de origem e destino existem no tráfego. Mais especificamente, cada fluxo é determinado pela 5-tupla *<IP de origem, porta de origem, IP de destino, porta de destino, protocolo (TCP/UDP)>*.

É importante observar que a quantidade de fluxos pode influenciar a capacidade das soluções de NIDS em processar os pacotes quando o tráfego avaliado é segregado por fluxo para balanceamento de carga entre as CPUs disponíveis.

A segunda abordagem diz respeito à quantidade de regras carregadas pelo NIDS. Como é de se esperar, quanto maior o número de regras, maior a taxa de pacotes que as soluções deixam de analisar. No entanto, observa-se que as soluções em espaço de kernel apresentam uma variação menor nessa taxa, enquanto o Snort e o Suricata mostram um aumento significativo nas suas taxas de perda de pacotes à medida que o número de regras cresce.

O Snort e o Suricata fornecem, ao final de suas execuções, o número de pacotes analisados, permitindo calcular quantos pacotes não foram avaliados, com base no total de pacotes enviados. Para o Sapo-boi e o PF IDS, foi adicionado um contador de pacotes ao Módulo de Suspeição (programa XDP) por meio de um mapa BPF, possibilitando a determinação do número de pacotes analisados por essas soluções. As Seções a seguir detalham os experimentos realizados para ambas as abordagens.

6.1.1 Número de Fluxos

Antes de analisar os resultados apresentados pelas soluções de NIDS em relação ao número de fluxos presentes no tráfego de rede, é importante compreender como os pacotes recebidos são segregados para processamento. Em placas de rede modernas, esse processo é realizado por meio do algoritmo RSS (*Receive Side Scaling*). O RSS detecta os pares IP-Porta e, com base em uma tabela *hash*, determina qual CPU será responsável por processar o pacote recebido Woo e Park (2012).

A Figura 6.1 apresenta a taxa de pacotes processados por CPU para tráfegos contendo 20 e 500 fluxos. Ambos os cenários de tráfego continham a mesma quantidade de pacotes, com o mesmo tamanho. Nota-se que, no caso de 500 fluxos, todas as CPUs recebem pacotes para processar; no entanto, com apenas 20 fluxos, as CPUs 0, 1, 2, 3, 8 e 13 permanecem ociosas. Como resultado, cada CPU ativa precisa processar uma quantidade maior de pacotes para que todos sejam avaliados.

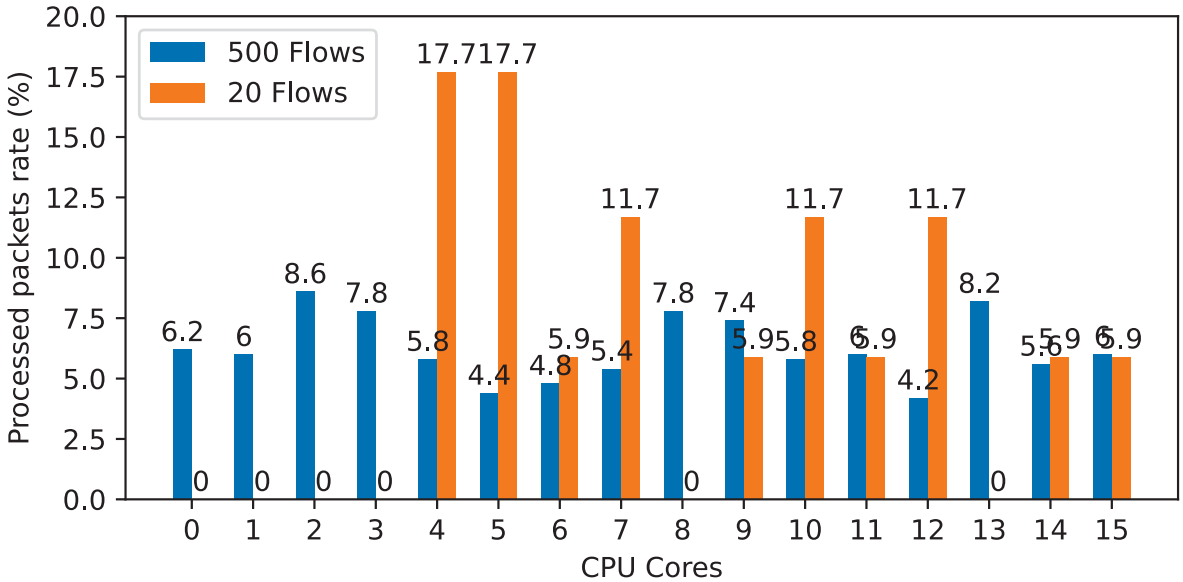


Figura 6.1: Taxa de pacotes avaliados por núcleo de CPU.

Com a intenção de avaliar como o número de fluxos afeta os IDS, é apresentada uma comparação das taxas de perda de pacotes das quatro soluções analisadas, todas carregadas com 13.000 regras. O tráfego enviado (3 milhões de pacotes, com 1.000 bytes de *payload*) é livre de pacotes maliciosos. O primeiro gráfico da Figura 6.2 exibe os resultados para tráfego com 20 fluxos, enquanto o segundo mostra o cenário com 500 fluxos.

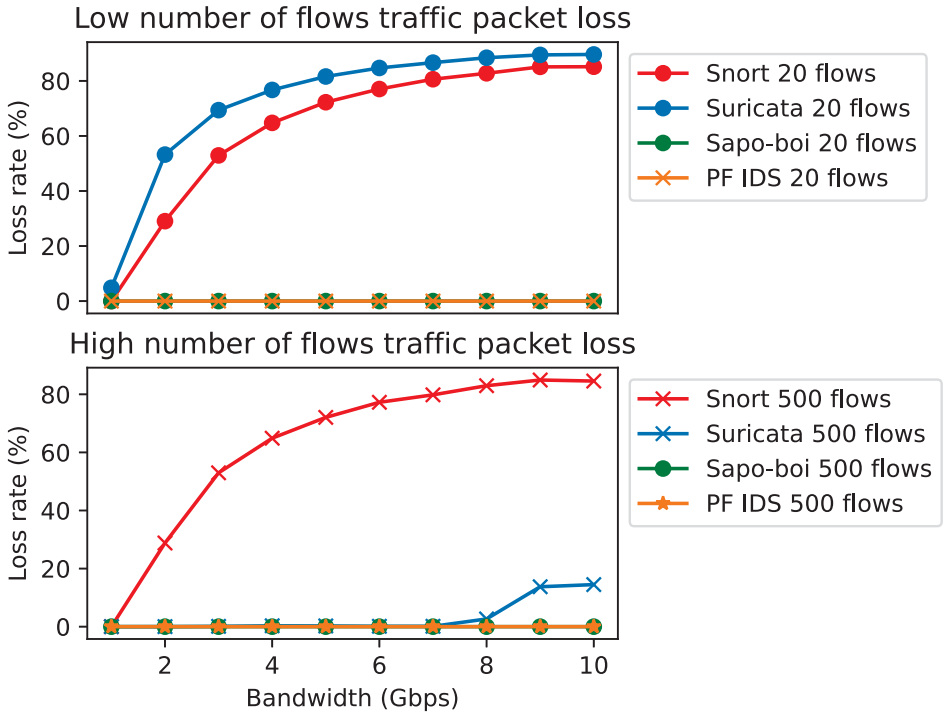


Figura 6.2: Taxa de perda de pacotes baseada no número de fluxos.

6.1.1.1 Suricata

O Suricata apresenta taxas de perda de pacotes menores à medida que o número de fluxos analisados pelo sistema aumenta. O bom desempenho do Suricata é atribuído à sua capacidade de utilizar o RSS como balanceador de carga, o que permite distribuir os fluxos entre as CPUs disponíveis. Isso garante uma distribuição equilibrada dos pacotes entre as CPUs, maximizando o número total de pacotes analisados.

Entretanto, quando o número de fluxos é reduzido, o Suricata sofre com a concentração do tráfego em um número menor de CPUs, o que aumenta as taxas de perda de pacotes.

Em outras palavras, o tráfego com alto número de fluxos beneficia o sistema. A 10 Gbps, com tráfego de baixo número de fluxos, o Suricata apresentou uma taxa de perda de pacotes de 89,61%. Entretanto, ao mudar para tráfego com alto número de fluxos, essa taxa caiu significativamente, para 14,68%.

Além disso, as taxas de perda permanecem próximas em 0% até 7 Gbps, aumentando para 2,65% a 8 Gbps, 13,74% a 9 Gbps e 14,68% a 10 Gbps. Este experimento destaca a importância de soluções IDS em espaço de usuário serem capazes de detectar e aproveitar a tecnologia RSS disponível nas placas de rede modernas para aquisição de pacotes. Também confirma a tendência esperada: quanto maior a largura de banda, maiores as taxas de perda de pacotes.

6.1.1.2 Snort

O Snort não é afetado significativamente por mudanças no número de fluxos, pois realiza a captura de pacotes em apenas um núcleo; embora o processamento dos pacotes, detalhado na Figura 3.2, ocorra em múltiplos núcleos, a captura do tráfego é restrita a um único *core*.

O sistema apresentou taxas de perda muito semelhantes em ambos os cenários, exibindo um comportamento oposto ao do Suricata e indicando que o número de fluxos não afeta a quantidade de pacotes analisados pelo sistema. Por exemplo, a 8 Gbps com tráfego de baixo número de fluxos, a taxa de perda foi de 82,76%, enquanto com tráfego de alto número de fluxos houve um leve aumento para 82,97%. Já a 10 Gbps, as taxas de perda foram de 85,15% e 84,57%, respectivamente. Esses resultados evidenciam que o Snort carece de mecanismos eficazes de balanceamento de carga na aquisição de dados entre os processadores.

6.1.1.3 Soluções em espaço de kernel

Para as soluções em kernel (Sapo-boi e PF IDS), a perda de pacotes permaneceu em zero para todas as larguras de banda, independentemente do número de fluxos. Isso ocorre porque não há tráfego malicioso, de modo que os pacotes são considerados seguros assim que o sistema reconhece que as portas de origem e destino não ativarão nenhum alerta. Como o XDP é capaz de processar até 24 milhões de pacotes por segundo em uma máquina com 6 núcleos Høiland-Jørgensen et al. (2018), esses resultados são esperados.

6.1.2 Número de Regras

A quantidade de regras carregadas por um NIDS, assim como a largura de banda na qual o tráfego é recebido, são os fatores mais críticos para a análise da perda de pacotes em soluções tradicionais de NIDS. Entretanto, sistemas baseados em kernel, devido à sua implementação, tendem a apresentar variações significativas nas taxas de perda principalmente em resposta às mudanças na largura de banda, enquanto mostram variação mínima em relação ao número de regras carregadas.

A Tabela 6.1 apresenta os resultados obtidos para diferentes sistemas, com variações nas taxas de transmissão e na quantidade de regras. O tráfego de alto número de fluxos (65535 fluxos) transmitido consiste em 3 milhões de pacotes, cada um com tamanho de payload de 1000 bytes, dos quais 5% eram maliciosos. Do total de pacotes, 90% eram TCP e 10% UDP. Cada dado na Tabela 6.1 representa a média e o desvio padrão de 10 execuções do NIDS analisado para cada combinação de número de regras e largura de banda.

6.1.2.1 Soluções em espaço de kernel

Observa-se que o Sapo-boi e o PF IDS apresentam uma baixa taxa de perda. Esse fenômeno pode ser atribuído à integração do IDS na pilha de rede do kernel, garantindo que, se um pacote for processado pela pilha, ele já tenha sido avaliado pelo IDS.

Mesmo que os programas XDP de ambas as soluções em kernel sejam diferentes com relação ao tratamento dos mapas e à maneira como os autômatos são construídos e avaliados em mapas BPF, ambas as soluções se aproveitam da verificação antecipada dos padrões, descartando precocemente os pacotes não suspeitos. Por isso, ao considerar o desvio padrão, as taxas de perda para ambos os sistemas são, na prática, equivalentes.

Outra observação importante é que ambas as soluções baseadas em kernel não deixam de avaliar nenhum pacote quando carregadas com uma única regra. Isso pode ser atribuído ao fato de que a regra carregada não correspondeu a nenhum par de portas no tráfego utilizado, o que

Tabela 6.1: Taxas de Perdas de Pacotes Referentes ao Número de Regras Carregadas.

Num. Regras	Largura de Banda	Sapo-boi	PF IDS	Snort	Suricata
1	1 Gbps	0% ± 0	0% ± 0	0% ± 0	0% ± 0
	2 Gbps	0% ± 0	0% ± 0	0% ± 0	0% ± 0
	4 Gbps	0% ± 0	0% ± 0	0,07% ± 0,009	0% ± 0
	8 Gbps	0% ± 0	0% ± 0	0,13% ± 0,03	2,31% ± 0,03
	9 Gbps	0% ± 0	0% ± 0	0,2% ± 0,01	2,39% ± 0,04
	10 Gbps	0% ± 0	0% ± 0	0,2% ± 0,09	2,41% ± 0,05
2000	1 Gbps	1,84% ± 0,03	1,83% ± 0,01	0% ± 0	0% ± 0
	2 Gbps	2,19% ± 0,002	2,19% ± 0,30	44,3% ± 0,004	0% ± 0
	4 Gbps	2,18% ± 0,003	2,19% ± 0,09	74,42% ± 0,005	0,28% ± 0,01
	8 Gbps	2,20% ± 0,02	2,20% ± 0,08	88,95% ± 0,03	2,99% ± 0,8
	9 Gbps	2,19% ± 0,02	2,19% ± 0,02	89,66% ± 0,05	3,55% ± 0,47
	10 Gbps	2,20% ± 0,02	2,20% ± 0,02	89.47% ± 0,06	3,83% ± 0,93
8000	1 Gbps	1,84% ± 0,04	1,84% ± 0,01	3,74% ± 0,009	0% ± 0
	2 Gbps	2,20% ± 0,03	2,19% ± 0,03	53,55% ± 0,01	0% ± 0
	4 Gbps	2,20% ± 0,03	2,23% ± 0,07	78,70% ± 0,005	0,49% ± 0,01
	8 Gbps	2,19% ± 0,02	2,19% ± 0,15	90,81% ± 0,02	4.17% ± 0,03
	9 Gbps	2,20% ± 0,02	2,25% ± 0,05	90,02% ± 0,05	5,24% ± 0,07
	10 Gbps	2,19% ± 0,01	2,22% ± 0,04	91,39% ± 0,04	6,22% ± 0,08
13000	1 Gbps	1,84% ± 0,03	1,84% ± 0,01	8,55% ± 0,01	0% ± 0
	2 Gbps	2,18% ± 0,02	2,22% ± 0,03	56,24% ± 0,009	0,002% ± 0,006
	4 Gbps	2,20% ± 0,01	2,20% ± 0,04	79,92% ± 0,004	0,61% ± 0,05
	8 Gbps	2,20% ± 0,02	2,20% ± 0,18	91,36% ± 0,02	5.49% ± 0,09
	9 Gbps	2,19% ± 0,01	2,18% ± 0,02	91,63% ± 0,05	6,77% ± 0,8
	10 Gbps	2,19% ± 0,03	2,22% ± 0,56	91,55% ± 0,05	7,88% ± 2,52
16000	1 Gbps	1,84% ± 0,03	1,85% ± 0,01	8,41% ± 0,01	0% ± 0
	2 Gbps	2,19% ± 0,01	2,18% ± 0,03	56,20% ± 0,01	0,012% ± 0,008
	4 Gbps	2,21% ± 0,03	2,18% ± 0,03	79,91% ± 0,004	0,73% ± 0,09
	8 Gbps	2,19% ± 0,01	2,22% ± 0,14	91,26% ± 0,02	7,88% ± 0,06
	9 Gbps	2,19% ± 0,01	2,20% ± 0,05	91,63% ± 0,05	8,60% ± 1,26
	10 Gbps	2,23% ± 0,01	2,22% ± 0,02	91,70% ± 0,05	9,61% ± 1,01

permite que o sistema encerre o processamento antes de executar qualquer rotina de casamento de padrões, conforme explicado no Capítulo 4 (para o Sapo-boi).

6.1.2.2 Sistemas em espaço de usuário

O Snort apresenta altas taxas de perda, deixando de analisar mais de 50% dos pacotes quando carregado com 8.000 regras a 2 Gbps, e mais de 90% a 8 Gbps. É importante destacar o aumento significativo na perda de pacotes entre 1 Gbps e 2 Gbps para todos os conjuntos de regras a partir de 2.000 regras, indicando que a taxa de chegada dos pacotes impacta fortemente a capacidade do Snort em analisá-los.

Embora não esteja mostrado na tabela, a taxa de perda do Snort a 2 Gbps foi de 2,31% com 200 regras. Contudo, essa taxa sobe para 26,71% com 500 regras na mesma largura de banda, evidenciando a sensibilidade do sistema ao aumento no número de regras carregadas.

O Snort analisa mais pacotes que o Suricata quando carregado com uma única regra. No entanto, à medida que o número de regras aumenta, fica evidente que o Suricata mantém taxas de perda baixas, enquanto o Snort começa a deixar de analisar um número progressivamente maior de pacotes. Por exemplo, o Suricata perde apenas 4,17% dos pacotes a 8 Gbps com 8.000

regras carregadas, enquanto o Snort perde mais de 90% nas mesmas condições. Esses resultados demonstram que o Suricata escala de forma mais eficaz que o Snort.

6.1.2.3 Comparação entre Sapo-boi e Suricata

Apesar do desempenho superior do Suricata com relação ao Snort, o primeiro ainda deixa de avaliar quantidade considerável de pacotes. Para efeito de comparação, Sapo-boi e Suricata foram testados com 16.000 regras, em larguras de banda que variam de 7 a 10 Gbps. Os resultados são mostrados na Figura 6.3.

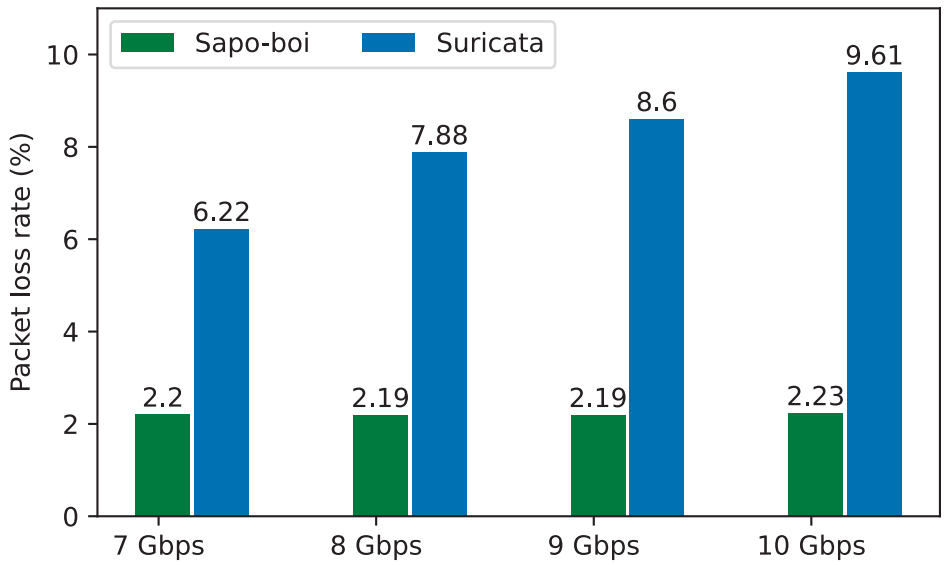


Figura 6.3: Sapo-boi e Suricata com 16,000 regras.

Observa-se que as taxas de perdas do Suricata aumentam significativamente, indicando que tanto o número de regras quanto a largura de banda afetam de fato a capacidade do sistema de avaliar pacotes. Em contraste, para o Sapo-boi, a taxa de perdas permanece relativamente estável até 10 Gbps.

A Figura 6.4 ilustra o ponto em que as taxas de perda entre o Sapo-boi e o Suricata se invertem. O PF IDS foi excluído desta análise para fins de visualização, uma vez que sua taxa de perda é semelhante à do Sapo-boi (considerando o desvio padrão). O Snort também foi deixado de fora devido à sua alta taxa de perda.

Os gráficos para 1, 4000 e 7000 regras mostram a superioridade do Suricata, com uma pequena discrepância entre as taxas em relação ao Sapo-boi. Com 8000 regras, as taxas de perda são semelhantes para ambos os sistemas até 5 Gbps. Contudo, após 9 Gbps, o Suricata apresenta taxas de perda mais elevadas. Os gráficos restantes destacam a crescente disparidade nas taxas de perda além de 9 Gbps, reforçando novamente a tendência de estabilização no Sapo-boi e o aumento nas perdas do Suricata. Considerando que um NIDS precisa tratar todo o tráfego do segmento de rede onde está inserido, as taxas de pacotes podem facilmente ultrapassar 8 Gbps. Portanto, o Sapo-boi é adequado para essa topologia.

Em resumo, quatro principais conclusões emergiram da análise:

- Snort e Suricata apresentam taxas de perda de pacotes diferentes. Enquanto o Snort deixa de analisar 5% dos pacotes a 1 Gbps com 8.000 regras, o Suricata apresenta uma taxa similar de perda a 9 Gbps com o mesmo número de regras, evidenciando o desempenho superior do Suricata entre os sistemas em espaço de usuário.

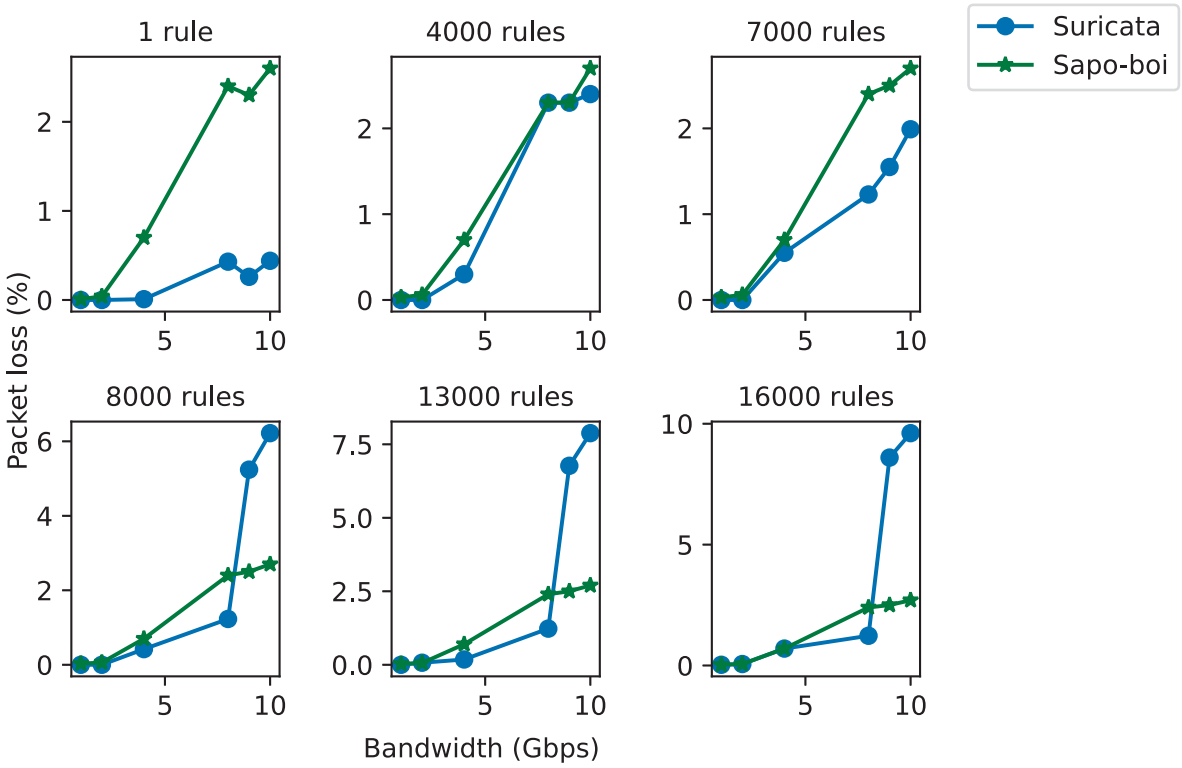


Figura 6.4: Reversão da taxa de perda de pacotes entre Suricata e Sapo-boi.

- As taxas de perda de pacotes em sistemas de espaço de usuário são fortemente influenciadas pelo número de regras carregadas, enquanto para soluções baseadas em kernel essa influência é reduzida.
- Ambas as soluções em espaço de *kernel* possuem desempenho equivalente, quando considerada a quantidade de pacotes avaliados.
- O Suricata apresenta excelente desempenho com tráfego de alto número de fluxos, quando carregado com um baixo número de regras. Entretanto, à medida que o número de regras aumenta, as taxas de perda de pacotes também sobem. Conforme mostrado na Figura 6.4, há sempre um ponto de inflexão onde o Sapo-boi supera o Suricata quando ambas as soluções operam com 8.000 ou mais regras carregadas.

6.2 USO DE CPU

A Tabela 6.2 apresenta o uso da CPU dos quatro sistemas avaliados para 1, 500, 8.000 e 16.000 regras. Os dados representam a média de 10 execuções para cada NIDS. O desvio padrão não foi incluído para manter a legibilidade da tabela. Além disso, o objetivo principal é mostrar as diferenças no uso de CPU entre as soluções, e não a variação entre execuções do mesmo NIDS.

Cada célula da tabela deve ser interpretada da seguinte forma:

```
<usuário>-<kernel>**<softirq>
```

Onde “usuário” se refere ao uso de CPU no contexto de usuário, “kernel” representa o tempo de CPU no contexto do kernel, e “softirq” indica o uso do processador pela *softirq*

NET_RX_SOFTIRQ, contexto no qual é executada a pilha de rede do *kernel* Linux, e portanto, os programas XDP.

Os dados da Tabela 6.2 representam o uso total de CPU somado entre todos os núcleos do processador. Como o *host* que executa os sistemas possui 16 núcleos, as taxas variam de 0% a 1600%. O uso de CPU nos espaços de usuário e kernel foi coletado por meio da ferramenta *pidstat*, enquanto o tempo de CPU no contexto *softirq* foi obtido utilizando a ferramenta *mpstat*. O tráfego utilizado nos testes é o mesmo descrito na Seção 6.1.2.

Tabela 6.2: Uso de CPU para as Soluções Avaliadas.

Num. Regras	Largura de Banda	Sapo-boi	PF IDS	Snort	Suricata
1	1 Gbps	0-0**3	0-0**2	480-582**514	143-75**117
	2 Gbps	0-0**2	0-0**2	436-636**634	272-172**270
	4 Gbps	0-0**7	0-0**9	604-784**725	230-168**256
	8 Gbps	0-0**27	0-0**26	606-940**897	325-240**351
	9 Gbps	0-0**28	0-0**26	584-978**939	393-280**390
	10 Gbps	0-0**29	0-0**32	572-1000**971	421-316**424
500	1 Gbps	2-9**24	1-6**6	1372-226**225	256-82**121
	2 Gbps	1-8**56	1-7**16	1278-303**328	312-128**192
	4 Gbps	2-13**74	2-6**35	1296-304**305	299-150**208
	8 Gbps	3-9**118	3-8**58	1010-528**532	565-263**344
	9 Gbps	4-10**121	4-8**66	1013-585**588	624-303**381
	10 Gbps	4-10**177	4-8**81	978-618**625	670-339**404
8000	1 Gbps	4-18**43	7-9**11	1432-168**177	376-88**122
	2 Gbps	4-21**70	11-12**25	1346-252**279	421-109**154
	4 Gbps	5-24**103	12-13**33	1327-271**275	388-131**178
	8 Gbps	5-32**230	13-11**110	1121-490**495	876-303**337
	9 Gbps	6-33**236	13-13**157	1057-541**544	952-354**377
	10 Gbps	8-46**240	16-20**160	1019-581**584	1019-385**396
16000	1 Gbps	2-17**45	6-8**11	1391-228**228	500-67**88
	2 Gbps	6-26**84	10-12**25	1274-322**301	477-87**109
	4 Gbps	6-25**107	10-12**42	1315-284**285	925-167**158
	8 Gbps	7-32**208	11-14**177	1088-515**521	1234-314**338
	9 Gbps	7-32**291	15-15**181	1021-577**574	1199-404**400
	10 Gbps	7-32**300	16-12**220	976-612**614	1146-379**389

6.2.1 Soluções em espaço de kernel

O primeiro ponto notável na tabela é a discrepância significativa entre os valores das soluções IDS baseadas em kernel (Sapo-boi e PF IDS) e aquelas em espaço de usuário (Snort e Suricata). Essa diferença pode ser explicada pelo fato de que os sistemas baseados em kernel descartam imediatamente os pacotes não suspeitos após a avaliação. Como apenas 5% do tráfego neste teste é malicioso, estes sistemas descartam 95% dos pacotes recebidos, evitando assim o processamento complexo normalmente realizado na pilha de rede do kernel, que é executado em contexto de *softirq*.

Também é importante destacar o uso zero de CPU nos contextos de usuário e kernel quando Sapo-boi e PF IDS são carregados com uma única regra. Esse comportamento ocorre porque nenhum pacote é encaminhado ao Módulo de Avaliação desses sistemas. Em outras palavras, não há operações envolvendo *sockets* XDP ou *perf events*, o que resulta em 0% de uso

no espaço de kernel. Além disso, como nenhum pacote é entregue ao Módulo de Avaliação, o uso do espaço de usuário também permanece em 0%.

O terceiro ponto que merece destaque é que o Sapo-boi utiliza mais CPU do que o PF IDS nos contextos de *kernel* e *softirq*, enquanto consome menos em espaço de usuário. Isso ocorre porque há um processamento adicional necessário nos contextos de kernel e *softirq* para realizar operações com *sockets* XDP em comparação aos *perf events*. Enquanto os *perf events* envolvem apenas a criação e o despacho de eventos, o uso correto e eficiente dos *sockets* XDP exige etapas mais complexas. Como detalhado nas Subseções 2.4 e 4.3.3, para cada pacote recebido pelo Módulo de Avaliação, diversas rotinas devem ser invocadas para manter corretamente as listas de endereços relativos utilizadas pelos módulos para escrita e leitura de pacotes.

O tempo de processamento ligeiramente maior no espaço de usuário para o PF IDS, quando mais regras são carregadas, pode ser atribuído ao fato de que os *perf events* são recebidos de forma que exige a extração dos dados do pacote a partir do evento. Esse processo demanda mais recursos do que o recebimento direto do pacote por meio do redirecionamento via *sockets* XDP. Além disso, como será discutido na Seção 6.3, as operações de *polling* para *perf events* são mais custosas que para *sockets* XDP.

Contudo, para ambas as soluções em kernel, o tempo em espaço de usuário é ínfimo quando comparado às soluções em espaço de usuário. Isso pode ser explicado por duas razões: (i) apenas 5% do tráfego foi enviado para processamento em espaço de usuário. Como havia 3 milhões de pacotes no tráfego, somente 150 mil deles teriam potencial para alcançar o espaço de usuário; (ii) as soluções são otimizadas para a busca rápida da regra a ser avaliada. Como o autômato de cada regra é pequeno, porque contém somente os padrões de uma regra, é esperado que o processamento em espaço de usuário não seja elevado.

6.2.2 Soluções em espaço de usuário

É importante destacar que o Suricata supera consistentemente o Snort na comparação entre os sistemas em espaço de usuário. O ponto de maior destaque é com 500 regras, onde a comparação mostra que o Snort usa significativamente mais CPU, em todos os contextos.

Para o Snort, a diferença entre o uso de CPU com 1 e 500 regras é notável. Para 1 regra e 10 Gbps, a solução usava em contexto de usuário, kernel e *softirq*, respesctivamente 572%, 1000% e 971%. Já para 500 regras, os índice são 978%, 618% e 625%.

Para o contexto de usuário, houve um aumento de uso de CPU de 406 pontos percentuais. Isso mostra a sensibilidade do Snort ao aumento de regras, evidenciando que o sistema passa mais tempo em espaço de usuário, onde ocorre a busca das assinaturas. Por passar mais tempo neste contexto, a solução não é capaz de realizar a aquisição de todos os pacotes que deveria (conforme verificado nos índices em contextos de kernel e *softirq*), corroborando com as altas taxas de perda exibidas da Seção 6.1.

Ademais, é interessante observar que para esses sistemas, as taxas de uso de CPU nos contextos de kernel e *softirq* são semelhantes. Essa similaridade ocorre porque a captura e cópia de pacotes, realizadas no kernel, acontecem de forma simultânea ao processamento da pilha de rede, que também é tratado no contexto *softirq*.

Em outras palavras, para o Snort e o Suricata, o uso de CPU em kernel representa a cópia dos pacotes durante o processamento da pilha de rede, enquanto o *softirq* mede o processamento completo dessa pilha, incluindo a própria cópia dos pacotes. A diferença entre os valores reportados se dá principalmente porque as ferramentas usadas para coletá-los são diferentes.

Tabela 6.3: Número de pacotes não-avaliados em espaço de usuário devido à falhas de comunicação entre módulos.

Bandwidth	Sapo-boi	PF-IDS
1 Gbps	0 ± 0	7218.3 ± 21148.45
2 Gbps	0 ± 0	744.0 ± 276.5
3 Gbps	28.9 ± 107.3	2446.5 ± 1451.8
4 Gbps	201.4 ± 115.3	2540.5 ± 354.2
5 Gbps	915.4 ± 314.1	18912.8 ± 9605.9
6 Gbps	3405.8 ± 1785.7	96907.8 ± 57453.6
7 Gbps	7214.5 ± 10207.1	79741.0 ± 78023.2
8 Gbps	5541.4 ± 5289.8	114803.2 ± 144410.1
9 Gbps	3895.4 ± 1084.3	59499.0 ± 51424.5
10 Gbps	4860.0 ± 3386.6	88207.1 ± 104650.8

6.3 TAXA DE PERDA NA COMUNICAÇÃO ENTRE MÓDULOS

Como as soluções IDS baseadas em kernel precisam encaminhar os pacotes suspeitos ao Módulo de Avaliação (em espaço de usuário) para veredito, é fundamental analisar a quantidade real de pacotes entregues ao processo em espaço de usuário. A Figura 6.5 ilustra o número de pacotes não recebidos em espaço de usuário ao longo de larguras de banda variando de 1 a 10 Gbps. Ambas as soluções foram carregadas com uma única regra, de forma que todo pacote recebido pelo módulo no kernel deve ser encaminhado ao espaço de usuário. No espaço de usuário, apenas um padrão deve ser buscado. Os testes foram realizados com 1 milhão de pacotes, cada um contendo 1000 bytes de *payload*. Os resultados apresentados correspondem à média de 10 execuções para cada largura de banda.

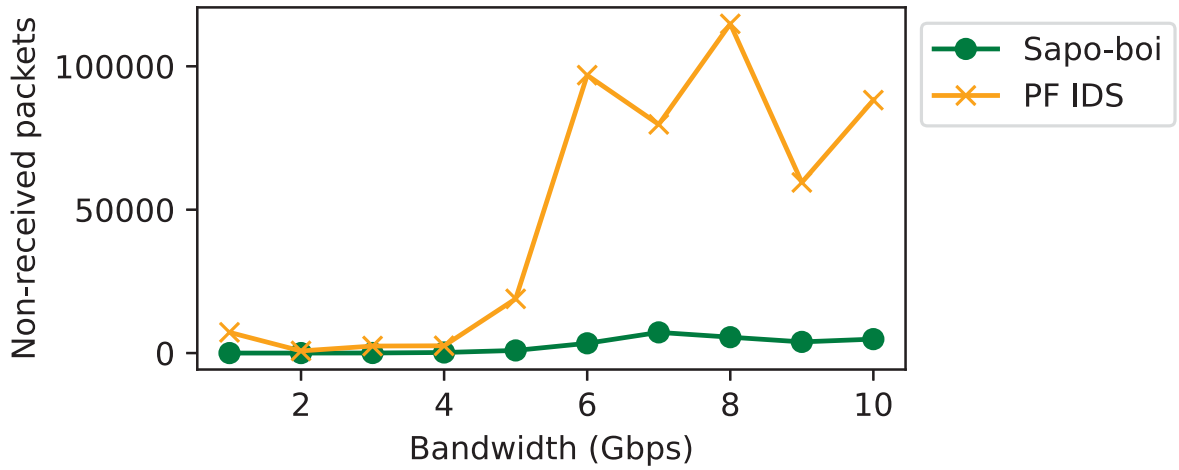


Figura 6.5: Taxas de perda de pacotes da comunicação entre os módulos para soluções em kernel.

A Figura 6.5 mostra que o Sapo-boi não é fortemente afetado pelo aumento da largura de banda ao se analisar o número de pacotes não recebidos. Em contraste, o PF IDS apresenta taxas de perda que variam de 5% a 11% após os 6 Gbps. Mais especificamente, a Tabela 6.3 apresenta a média e o desvio padrão de 10 execuções para cada uma das larguras de banda avaliadas. É possível observar que a média de pacotes não recebidos é maior para o PF IDS em todas as larguras de banda.

Em mais detalhes, a tabela mostra que há uma discrepância considerável entre os valores apresentados pelas duas soluções. Como exemplo, citam-se os valores para 7 Gbps, onde o Sapo-boi deixa de receber mais de 7000 pacotes, ao passo que para o PF IDS, esse valor sobe para quase 80000 pacotes.

As duas próximas seções buscam detalhar como funciona o processo de transferência de pacotes para ambas as soluções. Como os *perf events* representam maneira genérica de enviar eventos para o espaço de usuário, enquanto os *sockets* XDP são especializados para o redirecionamento de pacotes, os detalhes evidenciados pelas seções corroboram com os resultados apresentados na tabela.

6.3.1 PF IDS - *perf events*

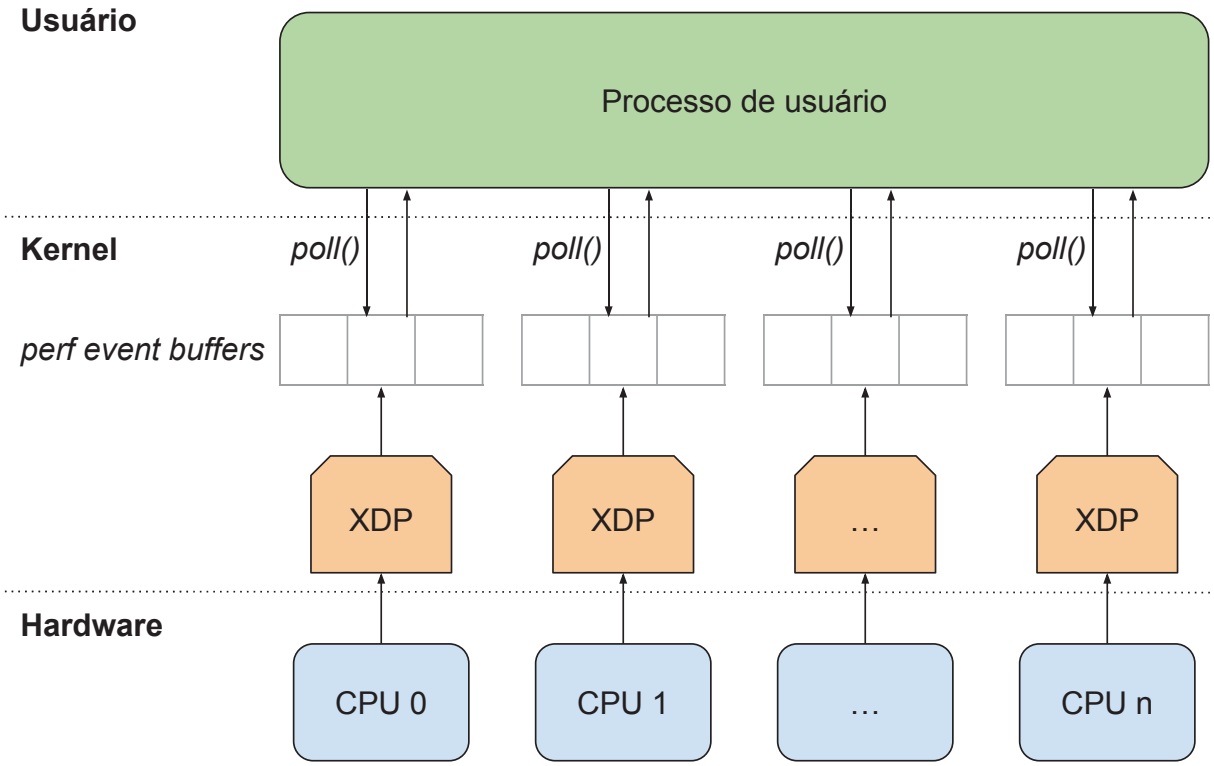


Figura 6.6: Aquisição de Pacotes - PF IDS

A Tabela 6.3 indica que o PF IDS deixa de entregar mais pacotes para a avaliação em espaço de usuário. Esta seção busca detalhar o processo de envio de *perf events*, a fim de evidenciar o porquê das diferenças tão significativas entre as soluções.

A Figura 6.6 mostra o envio de *perf events* para o espaço de usuário. Note que cada CPU executa um programa XDP, e cada programa XDP envia os eventos para um *buffer* de *perf events* específico. Observe também que essas estruturas estão localizadas em espaço de kernel.

O processo de usuário deve realizar chamadas à função *poll* para verificar se houveram novas escritas no *buffer*. Em caso positivo, o kernel deve verificar as permissões do processo e trocar a permissão dos eventos que estão no *perf buffer*. A seguir, os eventos daquele buffer poderão ser lidos.

Note que, como existe um *perf buffer* por CPU, este processo ocorre para todos *buffers*, toda vez que o processo de usuário verificar que houve uma escrita.

6.3.2 Sapo-boi - *sockets* XDP

A Figura 6.7 evidencia como os pacotes são transferidos para espaço de usuário através de *sockets* XDP, forma implementada pelo Sapo-boi para realizar a aquisição de pacotes.

Note que existe uma UMEM para cada *socket*, e que as UMEMs estão em espaço de usuário. Quando o programa XDP realiza o redirecionamento do pacote, o kernel fica responsável pela cópia dos pacotes para esta região.

Uma vez realizada a cópia, o kernel faz uma escrita no *RX ring* do *socket* XDP responsável pela operação, conforme detalhado na Seção 2.4. Portanto, para verificar se existem pacotes disponíveis em espaço de usuário, basta que o Módulo de avaliação verifique se há uma nova escrita no *RX ring* de qualquer socket. Em outras palavras, a função *poll()* no Sapo-boi consiste somente em uma verificação de escrita simples.

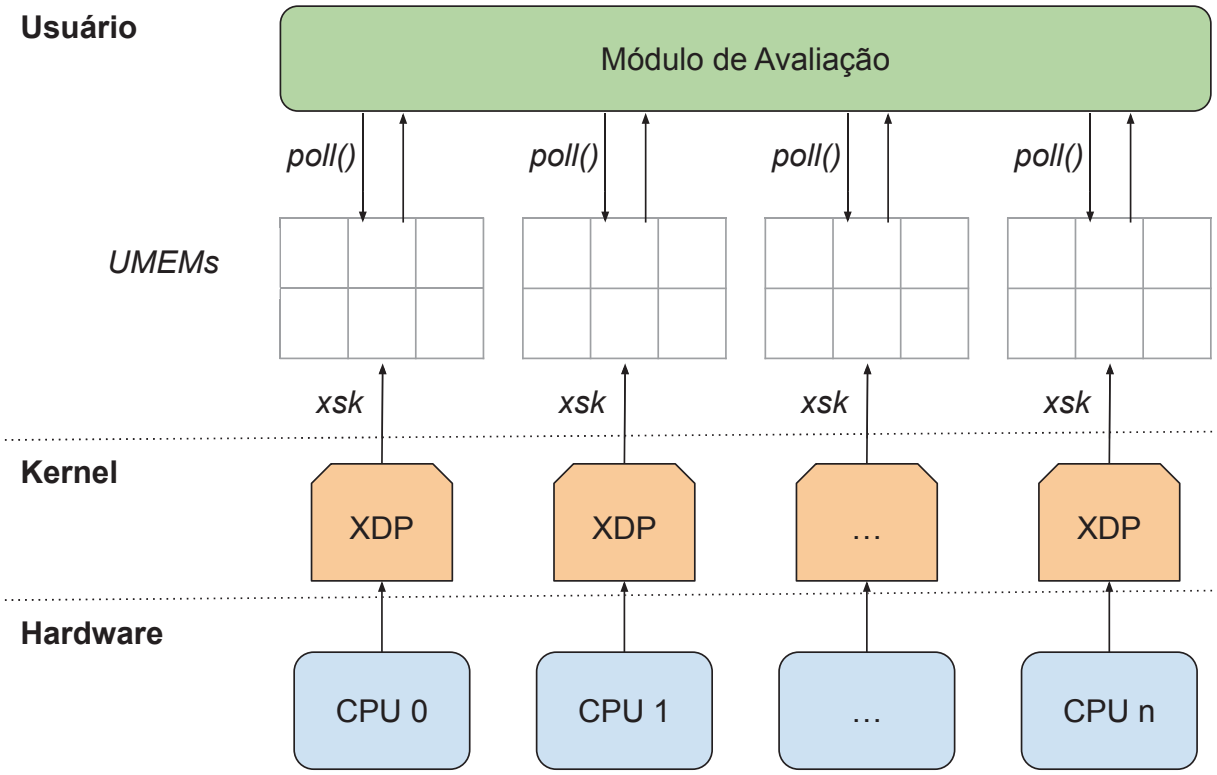


Figura 6.7: Aquisição de Pacotes - Sapo-boi

Se de fato houve uma escrita, significa que os pacote já foram transferido para o espaço de usuário, e que o Módulo de Avaliação já pode consumi-los, pois tem as permissões para fazê-lo. Ou seja, uma vez que o kernel realizou a escrita no *RX ring*, não existem mais verificações adicionais.

6.3.3 Comparação

A melhor performance do Sapo-boi pode ser explicada pelo fato de que a chamada '*poll*' para eventos em *sockets* XDP é mais rápida do que uma chamada '*poll*' para eventos em um *perf buffer*, estrutura utilizada pelo PF IDS como memória compartilhada para comunicação entre kernel e espaço de usuário.

Além disso, ao detectar eventos, o PF IDS deve solicitar ao kernel o direito de acessá-los, ao passo que, para o Sapo-boi, os pacotes já estão disponíveis para acesso, logo após a verificação através da função *poll()*.

6.3.4 Discussão - Perda de pacotes pelo Sapo-boi

Apesar de apresentar um número menor de pacotes não avaliados, o Sapo-boi ainda perde pacotes na comunicação entre módulos, mesmo usando *sockets* XDP. Karlsson e Töpel (2018) listam os principais problemas desse tipo de socket, expondo as razões pelas quais *drivers* em modo usuário, como o DPDK, ainda apresentam desempenho superior.

Para os experimentos desta dissertação, o ambiente usado nos testes não suportava o uso de *sockets* XDP no modo *ZERO_COPY*. Isso significa que, para cada pacote que deve ser redirecionado, há uma cópia de memória da região do *ring buffer* (onde está o pacote processado pelo programa XDP) para a área da UMEM, o que impacta negativamente o desempenho. Quando se usa o modo ZERO_COPY, a placa de rede pode escrever o pacote diretamente na UMEM, então o kernel precisaria apenas preencher alguns descritores nos *buffers* dos *sockets*, como explicado na Seção 2.4. No entanto, note que, ao usar um programa XDP, os *sockets* XDP continuam sendo a melhor escolha para encaminhar pacotes para o espaço de usuário, como corroborado na Tabela 6.3.

Em termos técnicos, pode-se afirmar que o Sapo-boi é mais robusto e confiável do que o PF IDS, pois um número maior de pacotes considerados suspeitos pelo Módulo de Suspeição é garantido de ser avaliado pelo Módulo de Avaliação. Isso também implica que a quantidade de pacotes maliciosos alertados pelo PF IDS pode ser menor do que a reportada pelo Sapo-boi, devido à possível perda de eventos durante a comunicação entre módulos.

7 CONCLUSÃO

Este trabalho propõe o Sapo-boi, um NIDS implementado em espaços de kernel e usuário capaz de realizar DPI para encontrar *fast patterns* no primeiro e demais padrões no segundo. Quando comparado com soluções em espaço de usuário (Snort e Suricata), é visível que o desempenho computacional da solução é superior, principalmente quando se refere à quantidade total de pacotes efetivamente avaliados pelas soluções.

De maneira geral, as soluções avaliadas que operam com programas XDP, ou seja, o Sapo-boi e o PF-IDS, possuem resultados semelhantes quando se compara o uso de CPU e a taxa de pacotes avaliados. Porém, quando se analisa a capacidade de passagem de pacotes do programa XDP para o espaço de usuário, a diferença entre as taxas de pacotes de fato entregues para o processo de usuário é notável, sendo significativamente maior para o Sapo-boi. Como discutido, isso reflete que a maneira escolhida para redirecionar pacotes influencia a robustez de uma solução NIDS baseada em programas XDP, haja vista que a proporção de pacotes enviados para avaliação em espaço de usuário é diretamente proporcional ao número de alertas que a solução pode gerar.

A estruturação da pilha de rede do kernel Linux foi explorada de forma a tirar vantagem do momento em que programas BPF na camada XDP são executados. Durante a arquitetura do sistema, percebeu-se que o fato deste tipo de programa ser executado antes de qualquer estrutura auxiliar de unidades de tráfego ser alocada pelo kernel é determinante para o bom desempenho da arquitetura proposta.

Além das limitações inerentes às soluções baseadas em assinatura, por exemplo, o fato de nenhuma assinatura não presente na base avaliada representar comportamento malicioso, o Sapo-boi ainda possui as limitações intrínsecas aos NIDS. Mais especificamente, a solução proposta, assim como os demais NIDS, que operam sobre espelhamento de tráfego, não é capaz de acionar controles preventivos, ou seja, são implementados de maneira a simplesmente detectar intrusões. Além disso, a solução não é capaz de detectar padrões maliciosos em pacotes cifrados.

Ademais, apesar do Sapo-boi ser capaz de superar as soluções em espaço de usuário, ele não conta com todas as ferramentas que compõem estas soluções. Mais especificamente, Snort e Suricata possuem pré-processadores capazes de remontar o tráfego recebido, além de manter estados sobre as sessões TCP e conversas UDP. Além disso, tais soluções ainda contêm módulos específicos para a geração de logs, além de implementar a ação de *reject*, que envia um pacote TCP *reset* ao detectar um padrão malicioso.

Como trabalhos futuros, propõe-se o uso do Sapo-boi como um sistema de detecção de intrusão em *hosts*, capaz de instrumentar funções usando o BPF para o processamento de tráfego depois da descriptografia. Ademais, é possível incrementar os testes usando outros modos de execução do XDP, mais notadamente o modo *driver* com *ZERO_COPY*, onde os dados do pacote são copiados diretamente para a UMEM, região de espaço de usuário, sem a necessidade de cópias adicionais; e o modo *offload*, no qual o programa XDP é descarregado diretamente uma uma *smartNIC*, e todo o processamento ocorre fora da CPU do *host*. Também é possível criar um mecanismo, baseado em mapas BPF, para manter informações (estados) sobre os fluxos TCP/UDP. Além disso, é possível estender o Módulo de Avaliação para a busca de protocolos de usuário, como DNS e HTTP. Por fim, outro possível trabalho poderia envolver o teste das ferramentas operando como sistemas de prevenção de intrusão, de modo a permanecer *inline* com os demais componentes de rede.

REFERÊNCIAS

Aho, A. V. e Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340.

Avast (2024). Avast blocks record breaking 10 billion attacks in 2023. https://press.avast.com/avast-blocks-record-breaking-10-billion-attacks-in-2023-nearly-a-50-increase-from-previous-year. Acessado em 14/03/2025.

Baidya, S., Chen, Y. e Levorato, M. (2018). ebpf-based content and computation-aware communication for real-time edge computing. Em *IEEE Conference on Computer Communications Workshops (INFOCOM)*, páginas 865–870.

Baykara, M. e Daş, R. (2019). Softswitch: A centralized honeypot-based security approach using software-defined switching for secure management of vlan networks. *Turkish Journal of Electrical Engineering and Computer Sciences*, 27(5):3309–3325.

Belson, D. (2025). Cloudflare 2024 year in review. https://blog.cloudflare.com/radar-2024-year-in-review/#:~:text=%2A%206.5,of%20traffic%20was%20mitigated. Acessado em 09/06/2025.

CNN (2024). Metade das empresas digitais no brasil perde até r$ 5 mi com fraudes, diz pesquisa. https://www.cnnbrasil.com.br/economia/macroeconomia/metade-das-empresas-digitais-no-brasil-perde-ate-r-5-mi-com-fraudes-diz-pesquisa/. Acessado em 13/03/2025.

D, S., Kataria, B., Sohoni, A. e Tahiliani, M. P. (2022). Implementation of nat44 and nat64 using tc-bpf and express data path (xdp). Em *2022 IEEE Bombay Section Signature Conference (IBSSC)*, páginas 1–6.

Firoz, N. F., Arefin, M. T. e Uddin, M. R. (2020). Performance optimization of layered signature based intrusion detection system using snort. Em *Cyber Security and Computer Science: Second EAI International Conference, ICONCS 2020, Dhaka, Bangladesh, February 15-16, 2020, Proceedings 2*, páginas 14–27. Springer.

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D. e Miller, D. (2018). The express data path: Fast programmable packet processing in the operating system kernel. Em *Proceedings of the 14th international conference on emerging networking experiments and technologies*, páginas 54–66.

Hu, Q., Asghar, M. R. e Brownlee, N. (2017). Evaluating network intrusion detection systems for high-speed networks. Em *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, páginas 1–6. IEEE.

Hu, Q., Yu, S.-Y. e Asghar, M. R. (2020). Analysing performance issues of open-source intrusion detection systems in high-speed networks. *Journal of Information Security and Applications*, 51:102426.

Hubballi, N. e Suryanarayanan, V. (2014). False alarm minimization techniques in signature-based intrusion detection systems: A survey. *Computer Communications*, 49:1–17.

Hubert, B. (2025). tc. `https://man7.org/linux/man-pages/man8/tc.8.html`. Acessado em 06/05/2025.

Jyothsna, V., Prasad, R. e Prasad, K. M. (2011). A review of anomaly based intrusion detection systems. *International Journal of Computer Applications*, 28(7):26–35.

Karlsson, M. e Töpel, B. (2018). The path to dpdk speeds for af xdp. Em *Linux Plumbers Conference*, volume 37, página 38.

Kostopoulos, S. (2024). *Machine learning-based near real time intrusion detection and prevention system using eBPF*. Bachelor's thesis, Hellenic Mediterranean University.

Liao, H.-J., Lin, C.-H. R., Lin, Y.-C. e Tung, K.-Y. (2013). Intrusion detection systems: A comprehensive review. *Journal of Network and Comp. Applications*, 36(1):16–24.

Lin, P.-C. e Lee, J.-H. (2013). Re-examining the performance bottleneck in a nids with detailed profiling. *Journal of Network and Computer Applications*, 36(2):768–780.

Lin, P.-C., Lin, Y.-D., Lai, Y.-C. e Lee, T.-H. (2008). Using string matching for deep packet inspection. *Computer*, 41(4):23–28.

Machnicki, R. K., Correia, J., Penteado, U., Fulber-Garcia, V. e Grégio, A. (2024). Sapo-boi: Pulando a pilha de rede no desenvolvimento de um nids baseado em bpf/xdp. Em *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, páginas 538–553. SBC.

Malek, Z. S., Trivedi, B. e Shah, A. (2020). User behavior pattern-signature based intrusion detection. Em *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, páginas 549–552. IEEE.

McCanne, S. e Jacobson, V. (1993). The bsd packet filter: A new architecture for user-level packet capture. Em *USENIX winter*, volume 46, páginas 259–270. Citeseer.

Otoum, Y. e Nayak, A. (2021). As-ids: Anomaly and signature based ids for the internet of things. *Journal of Network and Systems Management*, 29(3):23.

Ozkan-Okay, M., Samet, R., Aslan, Ö. e Gupta, D. (2021). A comprehensive systematic literature review on intrusion detection systems. *IEEE Access*, 9:157727–157760.

Pei, J., Hu, Y., Tian, L., Pei, X. e Wang, Z. (2025). Dynamic anomaly detection using in-band network telemetry and gcn for cloud–edge collaborative networks. *Computers & Security*, 154:104422.

Rizzo, L. (2012). netmap: a novel framework for fast packet i/o. Em *21st USENIX Security Symposium (USENIX Security 12)*, páginas 101–112.

Salah, K. e Kahtani, A. (2010). Performance evaluation comparison of snort nids under linux and windows server. *Journal of Network and Computer Applications*, 33(1):6–15.

Salim, J. H., Olsson, R. e Kuznetsov, A. (2001). Beyond softnet. Em *Annual Linux Showcase & Conference*.

SANTOS, R., JÚNIOR, E. P. C. e VIEIRA, L. F. (2019). Fast packet processing with ebpf and xdp: Concepts, code, challenges and applications.

Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K. e Carle, G. (2018). Performance implications of packet filtering with linux ebpf. Em *International Teletraffic Congress*, páginas 209–217.

Senado, A. (2024). Golpes digitais atingem 24% da população brasileira, revela datasenado. `https://www12.senado.leg.br/noticias/materias/2024/10/01/golpes-digitais-atingem-24-da-populacao-brasileira-revela-datasenado`. Acessado em 13/03/2025.

Shah, S. A. R. e Issac, B. (2018). Performance comparison of intrusion detection systems and application of machine learning to snort system. *Future Generation Computer Systems*, 80:157–170.

The Kernel Development Community (2025). sk_buff. `https://docs.kernel.org/networking/skbuff.html`. Acessado em 06/05/2025.

Uddin, M., Rahman, A. A., Uddin, N., Memon, J., Alsaqour, R. A. e Kazi, S. (2013). Signature-based multi-layer distributed intrusion detection system using mobile agents. *Int. J. Netw. Secur.*, 15(2):97–105.

Waleed, A., Jamali, A. F. e Masood, A. (2022). Which open-source ids? snort, suricata or zeek. *Computer Networks*, 213:109116.

Wang, H., Jha, S. e Ganapathy, V. (2006). Netspy: Automatic generation of spyware signatures for nids. Em *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, páginas 99–108. IEEE.

Wang, S.-Y. e Chang, J.-C. (2022). Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. *Journal of Network and Computer Applications*, 198:103283.

White, J. S., Fitzsimmons, T. e Matthews, J. N. (2013). Quantitative analysis of intrusion detection systems: Snort and suricata. Em *Cyber sensing 2013*, volume 8757, páginas 10–21. SPIE.

Woo, S. e Park, K. (2012). Scalable tcp session monitoring with symmetric receive-side scaling. *KAIST, Daejeon, Korea, Tech. Rep*, 144.

Xhonneux, M., Duchene, F. e Bonaventure, O. (2018). Leveraging ebpf for programmable network functions with ipv6 segment routing. Em *International Conference on emerging Networking EXperiments and Technologies*, páginas 67–72.

Zhengbing, H., Zhitang, L. e Junqi, W. (2008). A novel network intrusion detection system (nids) based on signatures search of data mining. Em *First International Workshop on Knowledge Discovery and Data Mining (WKDD 2008)*, páginas 10–16. IEEE.