SAPO-BOI: Pulando a Pilha de Rede no Desenvolvimento de um NIDS Baseado em BPF/XDP

Raphael Kaviak Machnicki[1], Jorge Correia[1], Ulisses Penteado[2], Vinicius Fulber-Garcia[1], André Grégio[1]

[1]Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

[2]BluePex – Centro de P&D
Limeira – SP – Brasil

{rkmachnicki,vinicius,gregio,jpcorreia}@inf.ufpr.br, ulisses@bluepex.com

Abstract. *Network Intrusion Detection Systems (NIDS) analyze different regions of a packet to detect known attack patterns. The advent of XDP has enabled the implementation of NIDS within the context of the Linux kernel network stack. In this work, we propose "SAPO-BOI," a NIDS composed of two modules: the Suspicion module (an XDP program that processes packets in parallel, discarding non-suspicious ones and redirecting suspicious packets for user-space decision) and the Evaluation module (a user-space process capable of finding the rule that analyzes the suspicious packet in constant time and generating alerts). Using a modified subset of Snort rules, SAPO-BOI was compared with traditional and kernel-level NIDS, outperforming the state-of-the-art.*

Resumo. *Sistemas de detecção de intrusão em redes (NIDS) analisam diferentes regiões de um pacote para detectar padrões de ataques conhecidos. O surgimento do XDP permitiu implementar NIDS no contexto da pilha de rede do kernel Linux. Neste trabalho, propõe-se o "SAPO-BOI", um NIDS composto por dois módulos: o de Suspeição (programa XDP que processa pacotes paralelamente, descarta os insuspeitos e redireciona os suspeitos para decisão em espaço de usuário) e o de Avaliação (processo de usuário capaz de encontrar em tempo constante a regra que analisa o pacote suspeito e gerar alertas). Usando um subconjunto modificado de regras do Snort, o SAPO-BOI foi comparado com NIDS tradicionais e em nível de kernel e superou o estado-da-arte.*

1. Introdução

O número de ameaças aos sistemas de computação conectados cresce na mesma proporção em que estes assumem um papel central na execução das mais variadas tarefas e na intermediação da comunicação dos usuários. Nesse contexto, soluções de segurança são protagonistas na detecção e mitigação das ameaças, a fim de garantir a confidencialidade, integridade e disponibilidade de sistemas e dados. Sistemas de Detecção de Intrusão baseados em Redes (*Network Intrusion Detection Systems* - NIDS) são soluções para identificar anomalias ou padrões de ataques conhecidos no tráfego de rede. Devido aos altos tráfegos de rede e à necessidade da aplicação de técnicas de Inspeção Profunda de Pacotes (*Deep Packet Inspection* - DPI), NIDS como Snort[1] e Suricata[2] não só apresentam

[1]https://www.snort.org
[2]https://suricata.io

baixo desempenho no processamento dos pacotes, mas também altas taxas de descarte [Alhomoud et al. 2011].

Uma alternativa é o uso de programas Linux baseados em *Berkeley Packet Filter* (BPF) [Graf et al. 2024], integrando-se à pilha de rede no *kernel* através do *eXpress Data Path* (XDP) [Høiland-Jørgensen et al. 2018]. O XDP permite que o programador tenha completo controle sobre os pacotes ingressantes em um sistema Linux, possibilitando a aceitação, rejeição, redirecionamento, reescrita e filtragem de tais pacotes. Assim, o XDP torna-se um gancho para programas BPF e uma ferramenta poderosa para o desenvolvimento de NIDS eficazes e eficientes.

Apesar do XDP ser popular para o monitoramento de redes e sistemas [Sundberg et al. 2023, Abranches et al. 2021], a tecnologia ainda é pouco explorada em aplicações de NIDS. Visando robustez e desempenho superiores aos NIDS tradicionais com as potenciais vantagens e recursos do XDP, este trabalho apresenta o Sistema de Avaliação e Processamento de tráfegO usando BPF e XDP para Observação de Intrusões (SAPO-BOI), um NIDS dividido em dois módulos: um em espaço de usuário (Módulo de Avaliação) e outro em espaço de *kernel* (Módulo de Suspeição). Este último é capaz de detectar padrões relacionados a ameaças em paralelo, logo na primeira camada da pilha de rede, e enviar pacotes suspeitos para o espaço de usuário através de *sockets* XDP.

Assim, este trabalho apresenta duas contribuições principais: (i) projeto e prototipação de um NIDS robusto com processamento de pacotes e redirecionamento destes por meio de *sockets* XDP, enviando metadados juntamente com o pacote para o espaço de usuário para auxiliar no processo de detecção de ataques; e (ii) a comparação de desempenho computacional da solução proposta com outros NIDS já existentes: duas delas executando inteiramente em espaço de usuário (Snort e Suricata) e uma mista, em espaço de *kernel* e usuário (descrita em [Wang and Chang 2022]). Os experimentos feitos mostraram que o SAPO-BOI supera as soluções inteiramente em espaço de usuário na capacidade de processamento de tráfego e empata com a solução em *kernel*, porém tem mais capacidade de passar alertas do espaço de *kernel* ao espaço de usuário, superando a solução implementada em *kernel*.

2. Fundamentação Teórica

Nesta seção, introduzem-se os principais conceitos que fundamentam o presente trabalho.

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 usam DPI 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 ataque que, se identificada em um pacote, gera um alerta. A Figura 1 ilustra uma regra simples suportada pelo Snort e Suricata (ambos NIDS por assinatura) e, antes do "->", mostra a ação a ser tomada no caso de ativação, bem como o protocolo a ser procurado, o IP e a porta de origem. Após o "->", há o IP e a porta de destino do tráfego analisado e, entre parênteses, as opções da regra (explicadas adiante).

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

Figura 1. Exemplo de regra de um NIDS.

As opções da regra contém, 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 mais representativo da regra (*fast_pattern*, pode existir zero ou um, definido pelo *content* imediatamente antes da palavra-chave) e um identificador único da regra (*sid*, exatamente um). Para executar a ação da regra, todos os padrões devem ser satisfeitos (operação de E lógico entre os *contents*). O processamento de regras para um pacote cessa após o primeiro *match*.

2.2. BPF e XDP

O BPF (*Berkeley Packet Filter*) é uma solução para processar eventos no nível de *kernel* por meio de *hooks* nativos que permite a implementação de programas em nível de usuário interpretados por uma máquina virtual BPF no *kernel*. Entretanto, devido ao modelo de operação e verificação de código dessa máquina virtual, várias restrições são aplicadas a um programa BPF (ex.: checagem de limites antes de todo acesso à memória). Isto provê uma camada extra de confiança, já que todo código executado foi verificado anteriormente [Vieira et al. 2020]. Com BPF, é possível estabelecer comunicação entre processos executando no espaço de *kernel* e no espaço de usuário, usando mapas (estruturas de dados chave-valor). Além disso, o BPF provê um conjunto de *helper functions*, responsáveis pela interação com o sistema ou com o contexto de execução das chamadas a partir de seus programas.

Já o XDP (*eXpress Data Path*) é um *hook* localizado na primeira camada da pilha de rede que opera no espaço do *driver*, sendo o primeiro ponto de contato entre a interface e o *kernel*. Programas BPF podem ser executados sobre o *hook* XDP e podem decidir o caminho de um quadro ou pacote recebido considerando, entre outras, as seguintes ações: 1) `XDP_PASS`: o pacote passa para a pilha de rede; 2) `XDP_DROP`: o pacote é descartado; 3) `XDP_REDIRECT`: o pacote é redirecionado para outra placa de rede ou para um programa de usuário. Antes de tomar qualquer uma dessas ações, o programa XDP pode acessar ou alterar o pacote.

A transmissão do tráfego de rede recebido por um programa XDP para um processo executando em espaço de usuário pode ser feita por meio de *sockets*. Os *sockets* da família XDP (AF_XDP ou XSK) executam essa tarefa mediante *bypass* das camadas superiores ao XDP na pilha de rede do *kernel*. Um *socket* AF_XDP é vinculado a uma única fila da placa de rede, mas uma fila pode ser associada a vários *sockets*. Além disso, ele é associado a uma única região de memória em espaço de usuário chamada UMEM (*Userspace Memory*), embora uma UMEM possa suportar vários *sockets* vinculados à mesma fila. A UMEM divide-se em *buffers* de tamanho homogêneo, nos quais é armazenado o tráfego entre o *kernel* e o espaço de usuário. O tamanho e a quantidade de *buffers* são configuráveis. A UMEM também possui o *fill ring*, que armazena endereços relativos de dados cuja propriedade foi transferida do espaço de usuário para o *kernel* e o *completion ring*, que mantém endereços relativos transferidos do *kernel* para o espaço de usuário. No contexto deste trabalho, “endereço relativo” é o deslocamento de memória usado para armazenar um pacote a partir do início do *buffer* referenciado por um *ring*.

Semelhantemente, um *socket* XDP possui os *rings* RX e TX, para pacotes recebidos e transmitidos, respectivamente. Quando um pacote é recebido, o *kernel* define um descritor no *ring* RX com o *buffer* da UMEM onde está o pacote, seu deslocamento no *buffer* e seu tamanho em bytes. Tanto os *rings* do *socket* XDP quanto os da UMEM operam sob política FIFO, com consumidores (TX e *completion*) e produtores (TX e *fill*) que permitem a troca de dados entre espaço de usuário e *kernel* ocorre via passagem de endereços de memória da UMEM entre eles (Figura 2). Os espaços de memória consumidos são re-disponibilizados aos produtores durante o ciclo de vida de um programa XDP.

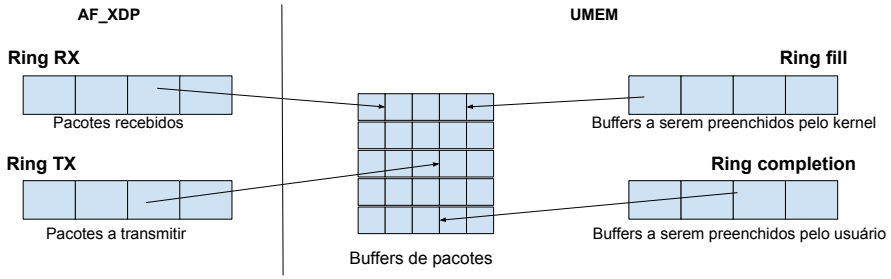

Figura 2. Funcionamento de *sockets* XDP.

Um recurso conveniente do BPF é o BTF (*BPF Type Format*) [Gregg 2019], que permite a definição de seções em programas BPF para descrever estruturas de dados em espaço de usuário. Tais estruturas são posteriormente reconhecidas em espaço de *kernel*, podendo ser alocadas no *kernel* e enviadas para o espaço de usuário via *sockets* XDP.

softirq e perf events. As *softirqs* consistem em operações de *software* executadas logo após uma interrupção de *hardware*. Para a rede, a chegada de tráfego na interface resulta em *irqs* (interrupções em *hardware*) seguidas de múltiplas *softirqs* que processam os pacotes através da pilha de rede do *kernel*. Por executar na primeira camada da pilha de rede, programas XDP estão no contexto das *softirq*. Portanto, para obter o tempo de CPU de um programa XDP, é necessário observar o tempo que o processador passa neste contexto. Já os *perf events* representam uma maneira genérica de enviar dados de um programa BPF para o espaço de usuário, cujo uso depende de se configurar um mapa BPF específico que proverá IPC (*Inter Process Communication*) entre os processos e dispensa a utilização de *sockets* XDP.

2.3. Algoritmo Aho-Corasick

O algoritmo Aho-Corasick é utilizado para buscar padrões em texto [Aho and Corasick 1975], sendo considerado especialmente eficiente na busca por um número arbitrário de *substrings* em uma dada *string*, como é o caso dos NIDS por assinaturas [Waleed et al. 2022]. O algoritmo tem como resultado um autômato preparado para reconhecer qualquer um dos padrões recebidos como entrada em uma *string* arbitrária, o qual pode ser construído sobre uma estrutura de dados tabular de chave-valor (e.g., mapas BPF). A complexidade do algoritmo na busca por padrões é $O(n + m + z)$, onde n é o tamanho da *string* em que se deseja encontrar os padrões, m indica o somatório dos tamanhos dos padrões, e z é o número de padrões na *string*. Assim, quanto menor o tamanho do autômato, melhor será o desempenho do algoritmo em termos de operações executadas em média.

3. Trabalhos Relacionados

Ataques contra redes ocorrem frequentemente e por diversos motivos, o que torna indispensável o uso de NIDS [Abhishta et al. 2020]. O Snort surgiu em 1998 para detectar padrões no *payload* de pacotes de rede, enquanto que o Suricata surgiu em 2010 com uma abordagem de detecção *multithread* para ser mais eficiente do que o primeiro. Diversos trabalhos comparam ambos, mas a maioria usa a versão *single-thread* do Snort [Murphy 2019, Park and Ahn 2017, White et al. 2013], o que resulta em melhor desempenho do Suricata (uso de CPU, memória e perda de pacotes). Trabalhos mais recentes, entretanto, mostram resultados que equiparam Suricata e Snort [Waleed et al. 2022, Hu et al. 2020], dado que o último passou a usar a mesma abordagem *multithread* em sua versão 3 (a partir de 2021). Embora ambos permitam o uso de BPF para filtragem de tráfego, nenhum é capaz de realizar verificações de padrões em contexto de *kernel*.

[Baidya et al. 2018] descreve um sistema BPF capaz de realizar DPI num contexto de *stream* de vídeos e Internet das Coisas. Contudo, o sistema funciona somente para pacotes específicos, o que não de adequa aos propósitos gerais de um NIDS. [Viljoen and Kicinski 2018] e [Xhonneux et al. 2018] demonstram que é possível realizar comutação e roteamento de pacotes em *kernel* com BPF, e até mesmo deslocar essas funcionalidades para coprocessadores executando em placas de rede inteligentes. O *In-Kev* [Ahmed et al. 2018] permite construir serviços de rede virtualizados usando chamadas consecutivas de programas BPF, incluindo funções de rede virtualizadas diretamente no *kernel*, em qualquer camada da pilha de rede. [Kostopoulos 2024] apresenta um NIDS 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.

[Wang and Chang 2022] introduz 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 suspeitos são transferidos do espaço de *kernel* para o de usuário. Uma característica importante desse programa, que o diferencia da proposta apresentada neste trabalho, é o modelo de comunicação e passagem de tráfego entre *kernel* e espaço de usuário, realizado por meio de *perf events*. Com isso, delimita-se o escopo de comparação entre a proposta deste trabalho e Snort, Suricata e PF IDS, tanto arquiteturalmente quanto por indicadores de desempenho, dado que esses NIDS fazem detecção por abuso.

4. Projeto e Implementação

A solução proposta neste trabalho, denominada SAPO-BOI[3] (**S**istema de **A**valiação e **P**rocessamento de tráfeg**O** usando **B**PF e XDP para **O**bservação de **I**ntrusões) é constituída por dois módulos principais: (i) Módulo de Suspeição, implementado como um programa XDP e executado ao nível de *kernel*, responsável por encontrar *fast patterns* das regras carregadas; e (ii) Módulo de Avaliação, um programa executado como um processo ao nível de usuário, cujo objetivo é definir se os demais padrões (*contents*) relacionados à regra referida pelo *fast pattern* previamente detectado pelo Módulo de Suspeição estão presentes no pacote marcado como suspeito.

[3]https://github.com/rkmach/SAPO-BOI

A Figura 3 mostra a arquitetura do SAPO-BOI em alto nível. Em sua fase de Inicialização (porção esquerda da figura), o Módulo de Avaliação carrega no *kernel* o Módulo de Suspeição e seus mapas BPF, representando autômatos *Aho-Corasick* criados a partir dos *fast patterns* das regras, bem como os seus próprios autômatos em espaço de usuário (estruturas de grafos que persistem até o fim da execução do IDS).

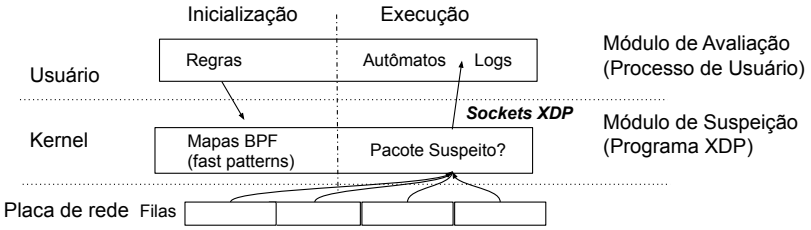

Figura 3. Visão geral da arquitetura do SAPO-BOI.

Após 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 na placa de rede 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 IDS 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 foi considerado insuspeito), 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 subseções a seguir detalham as operações realizadas por cada um dos módulos do SAPO-BOI.

4.1. 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 das unidades de tráfego (quadros e pacotes) são feitas pelo próprio Módulo (e.g., 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 cabeçalhos VLAN, o Módulo de Suspeição descartará esses cabeçalhos até a detecção do cabeçalho Ethernet correspondente. 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 concomitante-

mente. 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 and Park 2012].

Detecção de Padrões em Espaço de *Kernel*. Terminada a identificação do tráfego recebido, o Módulo de Suspeição verifica a existência dos *fast patterns* por meio de mapas BPF do tipo *hash*. Esses mapas permitem que sejam definidas estruturas abstratas como chave/valor, que representam estados de um autômato Aho-Corasick. Particularmente, as chaves representam um estado do autômato (inteiro) e uma transição (caractere), enquanto os valores definem os estados resultantes da transição e *flags* indicam se estes novos estados são finais ou não. 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.

Grupos de Portas TCP e UDP. 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. Logo, para criar autômatos enxutos, usa-se a estratégia de segmentação por grupos de portas TCP/UDP (cada grupo representa uma porta de origem e destino). 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. Assim, cada grupo de regras é mapeado para um único autômato. A Figura 4 exibe um exemplo da separação das regras em pares de portas. É possível observar que a regra com *signature id* (sid) 4 está 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. Esse fenômeno pode comprometer o desempenho do NIDS, pois o *fast pattern* correspondente ao sid 4 será testado para todo o tráfego de entrada.

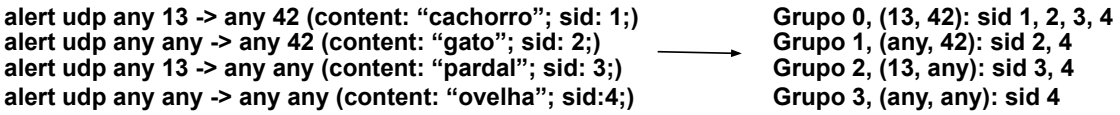

Figura 4. Agrupamento de regras por portas de origem e destino.

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 5. Cada entrada no mapa 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 que indica ao Módulo de Suspeição onde está o autômato de interesse. A partir desse autômato, se um *fast pattern* é encontrado no *payload* do pacote em análise, ele é enviado ao Módulo de Avaliação; caso contrário, o pacote é descartado.

Metadados. 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

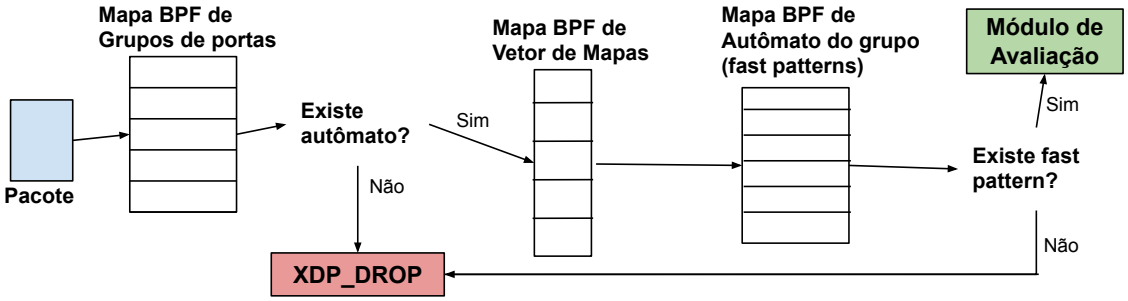

Figura 5. Caminho de um pacote no Módulo de Suspeição.

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 do *helper* `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 três 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; e (iii) uma sinalização que indica o protocolo da camada de transporte (TCP ou UDP). Essas informações permitem ao Módulo de Avaliação identificar eficientemente a regra que deve ser avaliada. Assim que todas as etapas representadas na Figura 5 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 o *helper* `bpf_redirect_map`.

4.2. 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, as quais serão detalhadas a seguir.

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 4.1, porém ainda não os inicializa). 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.

Processamento de 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 (Figura 5). O primeiro passo é determinar os grupos de portas (Figura 4), estruturas que contém, entre outros elementos, um vetor de regras. Os grupos em si também são armazenados em

um vetor específico, que diferenciam o tráfego pelo protocolo da camada de transporte. 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 6 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. Logo, quando o Módulo de Suspeição encontra um *fast pattern*, ele é capaz de enviar ao Módulo de Avaliação 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 Figuras 5 e 6). Dessa forma, um mesmo índice no vetor de mapas em *kernel*, representa o mapa que contém os *fast patterns* para aquele grupo de regras; enquanto 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.

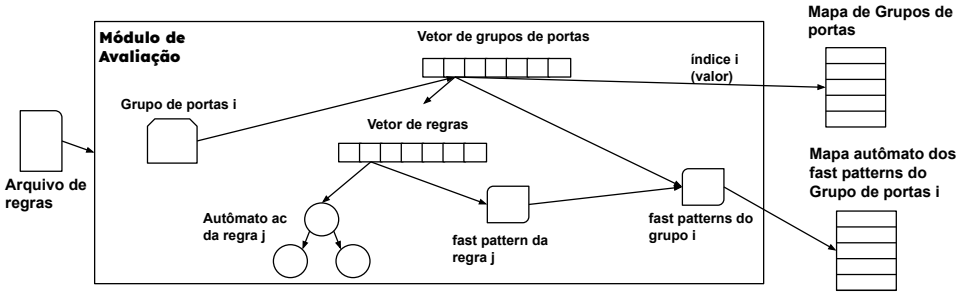

Figura 6. Estruturas do Módulo de Avaliação.

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.

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 2.2) 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`.

Gerenciamento e Processamento de Pacotes. Para receber pacotes, o Módulo de Avaliação executa uma função de *pool*, que recebe como argumento uma lista de descritores de *socket* e retorna o número de descritores onde há um evento. Assim, na ocorrência de eventos de interesse, verifica-se se houve alguma nova escrita no *ring* RX do *socket* correspondente. 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 considerando o seguinte *pipeline*: (i) Os metadados são processados e interpretados; (ii) Se o Módulo de Avaliação verifica que a regra não tem mais *contents* além do *fast pattern* previamente encontrado, a regra é aceita, um alerta é gerado e o processamento é encerrado; (iii) Caso contrário, o Módulo de Avaliação converte o *payload* do pacote em um vetor de caracteres, analisando-o por meio de um autômato da regra relacionada ao *fast pattern* detectado; (iv-i) Se o número total de padrões encontrados for igual ao número de padrões registrados na regra em análise, um alerta é gerado e o processamento encerrado; (iv-ii) Caso contrário, nenhum alerta é disparado e o processamento é encerrado.

5. Testes e Resultados

Esta seção apresenta e discute os resultados da análise de quatro NIDS: SAPO-BOI, desenvolvido para este trabalho; PF IDS, descrito em [Wang and Chang 2022] e implementado como parte deste trabalho; Snort 3.81.84; e Suricata 7.0.5. O ambiente dos experimentos consiste de um *host* IDS e um *host* remetente, sendo que este último gera e envia tráfego de rede em diferentes taxas de transmissão. Ambos possuem as mesmas especificações: SO *Ubuntu 24.04 LTS; Kernel 6.8.0-31-generic; x86_64*; Placa de Rede *Mellanox Technologies MT27500 [ConnectX-3] 10Gbps*; Processador *12th Gen Intel(R) Core(TM) i7-12700; 20 cores*; e 16GB de memória RAM. As máquinas são conectadas por um cabo SFP (*Small Form-factor Pluggable*) E124936-D de cobre bidirecional.

As regras usadas nos experimentos são o mesmo subconjunto das regras do *registered ruleset* do Snort [Roesch et al. 2024] usado por [Wang and Chang 2022], do qual removeu-se quatro tipos de regras: as sem a opção *content*; as que necessitam de *plugins* e/ou módulos do Snort; aquelas cujo *fast pattern* é menor que 13 *bytes* (para evitar *fast patterns* pequenos, que poderiam estar presentes em uma grande quantidade pacotes); as que contêm opções não suportadas. Os pré-processadores, *plugins* e módulos de ambos Snort e Suricata não envolvidos com detecção de padrões foram desabilitados, exceto

aqueles que mostram as estatísticas de processamento, ao final da execução do IDS. Snort e Suricata foram testados com seus respectivos modos "padrão" para captura de pacotes—a biblioteca *libdaq*, que provém uma camada de abstração para a *libpcap* para o primeiro, e *sockets AF_PACKET* para o último. Para gerar tráfego malicioso, foi desenvolvido um programa que analisa as regras estabelecidas e forja pacotes maliciosos utilizando as bibliotecas *scapy* e *pytbull*. O utilitário *iperf3* foi usado para gerar tráfego não-malicioso. Os cenários criados foram armazenados em arquivos *pcap* e reproduzidos no ambiente de testes com a ferramenta *tcpreplay* de acordo com uma taxa de envio pré-estabelecida.

5.1. Pacotes Não Analisados

Para verificar a quantidade de pacotes não analisados pelas soluções de IDS, dois cenários foram usados: (i) com variação do número de fluxos não maliciosos presentes no tráfego enviado e (ii) com variação do número de regras carregadas nas soluções. Por pacotes não analisados, compreendem-se pacotes que passam pelo *host* onde o IDS opera, porém, por motivos diversos, não são capturados e/ou analisados. Ou seja, define-se uma taxa de perda de pacotes de uma solução de IDS.

A Figura 7 apresenta a taxa de perda das soluções de IDS variando a taxa de envio de pacotes (largura de banda) e o número de fluxos em atividade, podendo ser tanto baixo (cenário "Baixo fluxo", com 2 fluxos apenas), quanto alto (cenário "Alto fluxo", com 1 milhão de fluxos). O número de regras carregadas (6 mil) é mantido estático.

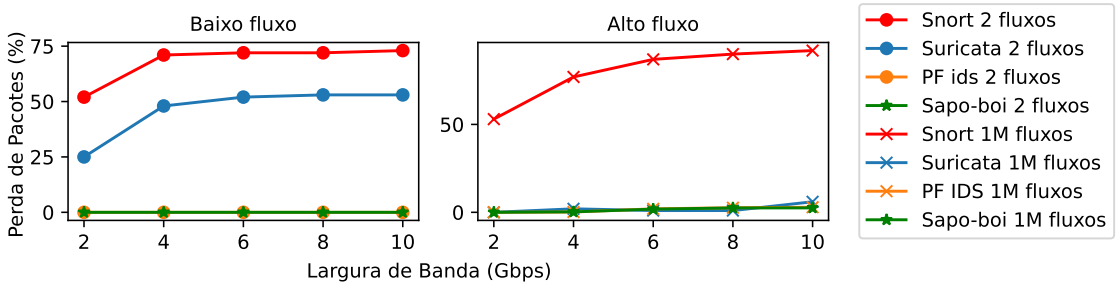

Figura 7. Perda de pacotes em relação ao número de fluxos.

É possível notar que o Suricata apresenta menores taxas de perda de pacotes conforme aumenta o número de fluxos que a solução analisa para uma mesma taxa de tráfego. Isso ocorre devido ao uso de um balanceador de carga, que distribui o processamento dos fluxos entre os núcleos disponíveis, ferramenta ausente no Snort. Particularmente, com um tráfego de 10Gbps, o Snort não analisa cerca de 90% dos pacotes, enquanto a taxa de perda do Suricata varia entre 56% (baixo fluxo) e 6% (alto fluxo).

A perda de pacotes nas soluções em *kernel* (SAPO-BOI e PF IDS) se manteve em zero para todas as larguras de banda quando o número de fluxos é baixo. Isso ocorre por duas razões: ausência de tráfego malicioso (os pacotes são descartados antecipadamente), e repetição de buscas no mapa BPF (mesmos fluxos, gerando uma alta taxa de *cache hits* no sistema). Quando o tráfego é de alto fluxo, as soluções em *kernel* apresentam um aumento discreto na taxa de perda de pacotes. A 10 Gbps, o SAPO-BOI e o PF IDS registraram, em média, uma perda de 2,5% devido às taxas maiores de *cache miss*, que ocorrem quando cada pacote ingressante possui um par de portas diferente. Portanto, valores diferentes dos mapas BPF precisam ser carregados na memória a cada pacote.

Cabe ressaltar que as soluções em *kernel* usam balanceadores de carga intrinsecamente, através da tecnologia RSS (*Receive Side Scalling*), suportada por placas de rede modernas, que não se baseiam somente nos fluxos para realizar o balanceamento.

A Tabela 1 mostra a taxa de perda de pacotes das soluções de IDS considerando variações de taxas de transmissão e número de regras. O tráfego enviado é composto de 2 milhões de fluxos e 3 milhões de pacotes de 1000 bytes de *payload*, dos quais 5% eram maliciosos (90% TCP e 10% UDP). Cada célula da Tabela 1 exibe a média e o desvio padrão da taxa de perda de pacotes medida para 10 execuções do respectivo teste.

Tabela 1. Taxas de perda de pacote para as IDS analisadas.

Num. Regras	Banda	SAPO-BOI	PF IDS	Snort	Suricata
1	1 Gbps	0,01% ± 0,01	0,01% ± 0,01	0% ± 0	0% ± 0
	2 Gbps	0,04% ± 0,03	0,06% ± 0,03	0% ± 0	0% ± 0
	9 Gbps	2,37% ± 0,08	2,42% ± 0,08	1,80% ± 0,12	0,26% ± 0,49
	10 Gbps	2,60% ± 0,21	2,42% ± 0,23	2,33% ± 0,08	0,44% ± 0,69
8000	1 Gbps	0,05% ± 0,008	0,01% ± 0,01	5,50% ± 0,37	0% ± 0
	2 Gbps	0,04% ± 0,03	0,04% ± 0,3	54,05% ± 0,48	0% ± 0
	9 Gbps	2,50% ± 0,01	2,35% ± 0,05	92,12% ± 0,01	5,03% ± 0,41
	10 Gbps	2,70% ± 0,06	2,62% ± 0,14	92,44% ± 0,01	10,55% ± 0,43
16000	1 Gbps	0,04% ± 0,04	0,02% ± 0,01	5,51% ± 0,72	0% ± 0
	2 Gbps	0,03% ± 0,03	0,04% ± 0,03	54,12% ± 0,66	0,12% ± 0,01
	9 Gbps	2,59% ± 0,08	2,50% ± 0,05	91,86% ± 0,31	40,13% ± 0,38
	10 Gbps	2,52% ± 0,10	2,72% ± 0,02	92,36% ± 0,01	46,73% ± 0,86

Observa-se que SAPO-BOI e PF IDS apresentam uma taxa de perda baixa em todos os cenários pelo seu modelo de verificação de *fast patterns* em *kernel*. As pequenas variações entre as taxas de perda do SAPO-BOI e PF IDS são explicadas pelas diferentes estratégias de encaminhamento de pacotes suspeitos para o Módulo de Avaliação, porém estas não representam diferença estatística relevante.

O Snort apresenta taxas elevadas de perda, deixando de avaliar mais de 50% dos pacotes com 8000 regras e 2 Gbps de tráfego, e 92% a 9 Gbps. Em especial, é evidente a diferença na perda quando a taxa de tráfego varia entre 1 Gbps e 2 Gbps, sinalizando que esta é determinante para a estabelecer taxa de perda conforme o número de regras aumenta no Snort.

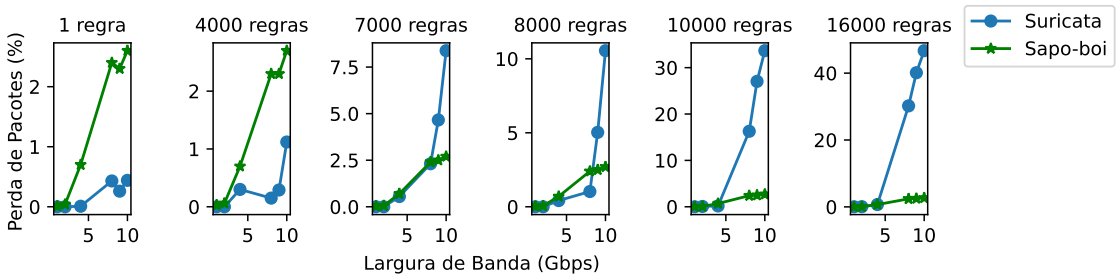

Figura 8. Inversão da taxa de perda entre SAPO-BOI e Suricata.

O Suricata, por sua vez, não apresenta taxa de perda de pacotes em larguras de banda de até 2 Gbps e com um número de regras menor ou igual a 8000. Mesmo com altas taxas de tráfego, o Suricata mantém taxas de perda abaixo de 11% para até 8000 regras, devido à sua eficiente capacidade de balanceamento de carga via fluxos. No entanto, quando o número de regras excede 8000, a taxa de perda do Suricata aumenta significativamente dada a progressão da taxa de tráfego, atingindo até 46% com 16000 regras e

10 Gbps. Isso revela uma limitação de desempenho do Suricata no ambiente de testes utilizado, apesar de sua boa capacidade de paralelização e balanceamento de carga.

A Figura 8 mostra o momento em que ocorre a inversão das taxas de perda entre SAPO-BOI e Suricata: para 1 e 4000 regras, Suricata é melhor, porém com pequena diferença entre as taxas de perda (menos de 3%); com 7000 regras e até 8 Gbps, as taxas se igualam para as duas soluções e, a partir de 9 Gbps, Suricata apresenta taxas de perda superiores às do SAPO-BOI. Para os casos com 7000 regras ou mais, sempre há um ponto de inflexão onde o Suricata apresenta maiores taxas de perda de pacotes do que o SAPO-BOI, conforme evolução da taxa de tráfego incidente no sistema.

5.2. Uso de CPU

A Tabela 2 exibe a média (5 execuções) do uso de CPU das quatro soluções de IDS avaliadas. O desvio padrão não foi informado para manter a tabela legível. Uma célula da tabela deve ser lida da maneira representada a seguir, onde <usuário> é o uso de CPU em contexto de usuário, <kernel> representa o tempo de CPU em *kernel*, e <softirq> o uso em contexto de *softirq*: `<usuário>-<kernel>**<sofirq>`.

Os dados representam o somatório do uso de CPU de todos os núcleos de processamento utilizados. Uma vez que o *host* que executa as soluções de IDS possui 20 núcleos, a taxa de uso pode variar entre 0% e 2000%. O uso de CPU nos espaços de usuário e *kernel* foi coletado pelo utilitário *pidstat*, enquanto o tempo de CPU em contexto de *softirq* foi coletado com a ferramenta *mpstat*. O tráfego usado para os testes foi o mesmo descrito para os resultados reportados na Tabela 1.

Tabela 2. Taxa de uso de CPU para as soluções de IDS analisadas.

Num. Regras	Banda	SAPO-BOI	PF IDS	Snort	Suricata
1	1 Gbps	0-0**2	0-0**2	600-728**643	179-94**147
	2 Gbps	0-0**2	0-0**2	546-796**793	314-216**338
	9 Gbps	0-0**28	0-0**28	730-1223**1174	492-350**488
	10 Gbps	0-0**29	0-0**32	716-1251**1214	527-396**531
8000	1 Gbps	4-18**43	7-9**11	1789-211**222	470-111**153
	2 Gbps	4-21**70	11-12**25	1683-315**349	500-137**193
	9 Gbps	6-33**236	13-13**157	1322-677**680	1190-443**472
	10 Gbps	8-36**240	16-20**160	1274-729**731	1247-482**495
16000	1 Gbps	2-17**45	6-8**11	1739-258**258	626-84**108
	2 Gbps	6-26**84	10-12**25	1593-403**403	599-104**137
	9 Gbps	7-32**291	15-15**181	1277-722**722	1498-465**467
	10 Gbps	7-32**300	16-17**220	1221-768**768	1433-474**487

A diferença entre os resultados das soluções híbridas em espaço de *kernel* e usuário (SAPO-BOI e PF IDS) em relação àquelas que executam exclusivamente em espaço de usuário (Snort e Suricata) é explicada pelo fato das soluções em *kernel* descartarem os pacotes não suspeitos imediatamente após o veredito. Como somente 5% do tráfego para este teste é malicioso, as soluções podem descartar 95% dos pacotes ingressantes, o que evita o processamento complexo que ocorre na pilha de rede do *kernel*, que executa em contexto de *softirq*. Também é necessário destacar o uso nulo de CPU em contextos de usuário e *kernel* quando SAPO-BOI e PF IDS estão carregados com 1 regra. Isso acontece porque nenhum pacote é enviado para o Módulo de Avaliação das soluções.

Outro ponto importante é que o SAPO-BOI usa mais CPU que PF IDS em contexto de *kernel* e *softirq*, ao passo que usa menos em contexto de usuário. Isso acontece, pois as operações de envio de tráfego via *sockets* XDP são mais custosas nesses contextos

do que as equivalentes em *perf events*. O tempo de processamento superior em espaço de usuário ser levemente maior para PF IDS quando há mais regras carregadas, é explicado pela necessidade de extração dos dados dos pacotes a partir de *perf events*, o que é dispensável quando o tráfego é encaminhado via *sockets* XDP.

Por fim, evidencia-se que o Suricata apresenta melhor desempenho, de maneira geral, em comparação ao Snort. Dentre as soluções executando exclusivamente em espaço de usuário, a taxa de uso e de CPU em *kernel* e *softirq* são semelhantes. Isso acontece porque a captura e cópia dos pacotes, em *kernel*, acontece juntamente com o processamento da pilha de rede, que também ocorre em *kernel*, em contexto de *sofirq*.

5.3. Perda de Pacotes na Comunicação Entre Módulos

Como as soluções em *kernel* devem enviar os pacotes suspeitos para uma análise aprofundada no Módulo de Suspeição, é importante calcular qual a taxa de pacotes encaminhados que são, de fato, entregues para a aplicação em espaço de usuário. A Figura 9 mostra a quantidade de eventos perdidos com taxa de tráfego variando entre 1 e 10 Gbps, para 8000 regras carregadas. O tráfego usado para os testes é o mesmo dos resultados da Tabela 1.

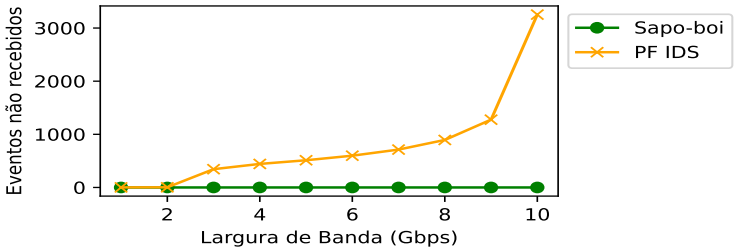

Figura 9. Perda de eventos das abordagens em *kernel*.

Observa-se que, para até 2 Gbps de tráfego, tanto SAPO-BOI quanto PF IDS foram capazes de entregar todos os eventos ao Módulo de Avaliação. A partir de 3 Gbps, o PF IDS passa a perder eventos, aumentando a taxa de perda ao passo que a taxa de tráfego cresce. Por outro lado, o SAPO-BOI é capaz de entregar todos os pacotes para avaliação em todos os casos. Tal evento acontece já que o SAPO-BOI realiza verdadeiramente o redirecionamento do pacote para a aplicação de usuário via *sockets*. O PF IDS, por sua vez, usa *perf events* que podem ser ignorados ou perdidos pela aplicação em espaço de usuário. Na prática, esse resultado demonstra a robustez do SAPO-BOI em avaliar tráfego de rede e buscar por padrões maliciosos no contexto dos testes realizados.

6. Conclusão

A eficiência e a eficácia dos NIDS são de grande impacto para uma sociedade cada vez mais conectada, e gera quantidades massivas de tráfego de rede. Neste trabalho, foi proposto o SAPO-BOI, um NIDS BPF por assinatura focado na minimização da perda de pacotes e alertas que superou os resultados obtidos por soluções de mercado como Snort e Suricata, além de se mostrar competitivo (superando em aspectos específicos) em relação a uma proposta de NIDS BPF do estado da arte. Trabalhos futuros incluem paralelizar o Módulo de Avaliação com *threads* individualizadas para filas específica das placas de rede dos sistemas, aumentando a capacidade de processamento do tráfego suspeito, bem como ampliar os cenários de testes para avaliar o NIDS em produção.

7. Agradecimentos

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.

Referências

Abhishta, A., van Heeswijk, W., Junger, M., Nieuwenhuis, L. J., and Joosten, R. (2020). Why would we get attacked? an analysis of attacker's aims behind ddos attacks. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 11(2):3–22.

Abranches, M., Michel, O., Keller, E., and Schmid, S. (2021). Efficient network monitoring applications in the kernel with ebpf and xdp. In *IEEE Conference on Network Function Virtualization and Software Defined Networks*, pages 28–34.

Ahmed, Z., Alizai, M. H., and Syed, A. A. (2018). Inkev: In-kernel distributed network virtualization for dcn. *ACM SIGCOMM Computer Communication Review*, 46(3):1–6.

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

Alhomoud, A., Munir, R., Disso, J. P., Awan, I., and Al-Dhelaan, A. (2011). Performance evaluation study of intrusion detection systems. *Procedia Computer Science*, 5.

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

Graf, T., Venugopalan, R., et al. (2024). Extended berkeley packet filter. `https://ebpf.io`. Acessado em: 25/05/2024.

Gregg, B. (2019). *BPF performance tools*. Addison-Wesley Professional.

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. (2018). The express data path: Fast programmable packet processing in the operating system kernel. In *International Conference on Emerging Networking Experiments and Technologies*, pages 54–66.

Hu, Q., Yu, S.-Y., and 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.

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

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

Murphy, B. R. (2019). *Comparing the performance of intrusion detection systems: Snort and Suricata*. PhD thesis, Colorado Technical University.

Park, W. and Ahn, S. (2017). Performance comparison and detection analysis in snort and suricata environment. *Wireless Personal Communications*, 94:241–252.

Roesch, M., Henderson, A., et al. (2024). Snort - open source intrusion prevention system. `https://www.snort.org`. Acessado em 16/05/2024.

Sundberg, S., Brunstrom, A., Ferlin-Reiter, S., Høiland-Jørgensen, T., and Brouer, J. D. (2023). Efficient continuous latency monitoring with ebpf. In *International Conference on Passive and Active Network Measurement*, pages 191–208.

Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys*, 53(1):1–36.

Viljoen, N. and Kicinski, J. (2018). Using ebpf as an abstraction for switching. *URL http://vger. kernel. org/lpc_net2018_talks/eBPF_For_ Switches. pdf*.

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

Wang, S.-Y. and 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., and Matthews, J. N. (2013). Quantitative analysis of intrusion detection systems: Snort and suricata. In *Cyber sensing*, volume 8757.

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

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