DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Defesa de borda orientada por SDN: Desafios de Sincronização e Persistência de Regras na Mitigação de Ameaças do Tipo Mirai

Clayton Menezes Silva

Brasília, 31 de Julho de 2026

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Defesa de borda orientada por SDN: Desafios de Sincronização e Persistência de Regras na Mitigação de Ameaças do Tipo Mirai

Clayton Menezes Silva

ORIENTADOR: PROFESSOR DR. FÁBIO LÚCIO LOPES DE MENDONÇA

DISSERTAÇÃO DE MESTRADO PROFISSIONAL EM ENGENHARIA ELÉTRICA

BRASÍLIA/DF, 31 de Julho de 2026

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Defesa de borda orientada por SDN: Desafios de Sincronização e Persistência de Regras na Mitigação de Ameaças do Tipo Mirai

Clayton Menezes Silva

Dissertação de Mestrado Profissional submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Dr. Fabio Lucio Lopes de Mendonça, FT/UnB _____
Presidente - Orientador

Prof. Dr. Daniel Alves da Silva, PPEE/UnB _____
Examinador Interno

Prof. Dr. Francisco Lopes de Caldas Filho, IESB - Instituto de Educação Superior de Brasília _____
Examinador Externo

Prof. Dr. João José Costa Gondim, FT/ENE/UnB _____
Membro Suplente

FICHA CATALOGRÁFICA

SILVA, CLAYTON MENEZES
Defesa de borda orientada por SDN: Desafios de Sincronização e Persistência de Regras na Mitigação de Ameaças do Tipo Mirai [Distrito Federal] 2026.
xi, 126 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2026).
Dissertação de Mestrado Profissional - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Redes Definidas por Software (SDN)	2. Segurança IoT
3. Botnet Mirai	4. Persistência de Regras
5. Hardware-in-the-Loop	
I. ENE/FT/UnB	II. PPEE/UnB

REFERÊNCIA BIBLIOGRÁFICA

SILVA, C. M. (2026). *Defesa de borda orientada por SDN: Desafios de Sincronização e Persistência de Regras na Mitigação de Ameaças do Tipo Mirai*. Dissertação de Mestrado Profissional, Publicação: PPEE.MP.118, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 126 p.

CESSÃO DE DIREITOS

AUTOR: Clayton Menezes Silva
TÍTULO: Defesa de borda orientada por SDN: Desafios de Sincronização e Persistência de Regras na Mitigação de Ameaças do Tipo Mirai.
GRAU: Mestre em Engenharia Elétrica ANO: 2026

Clayton Menezes Silva
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA

Dedico este trabalho, com todo o meu amor e gratidão, à minha mãe, Maria do Socorro Menezes da Silva. O seu exemplo de resiliência e a sua fé inabalável na minha capacidade foram o combustível que me impediu de desistir nos momentos de maior exaustão.

À minha esposa, Joana Maria de Bessa, e ao meu filho, Raphael Bessa Menezes, que são o meu norte e o meu propósito. Esta conquista carrega o peso do sacrifício e da compreensão de vocês perante as inúmeras horas em que o meu silêncio e a minha ausência foram necessários para a conclusão deste sonho. Esta vitória é, acima de tudo, de vocês.

Às minhas irmãs, cujo incentivo e presença constante foram alicerces fundamentais durante toda a caminhada deste mestrado.

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, por me conceder a vida e a força necessária para superar os desafios desta caminhada.

Ao meu orientador, Prof. Dr. Fábio Lúcio Lopes de Mendonça, pela orientação segura, pela amizade e, acima de tudo, pela paciência e disponibilidade no esclarecimento de dúvidas e na condução dos problemas complexos que surgiram durante o desenvolvimento desta dissertação.

Aos professores do Programa de Pós-Graduação Profissional em Engenharia Elétrica da Universidade de Brasília (PPEE/UnB), em especial aos professores Georges D. Amvame Nze e Francisco Lopes de Caldas Filho, pelos ensinamentos compartilhados, pelo apoio constante e pela amizade, fundamentais para o meu crescimento acadêmico e profissional.

Ao meu parceiro do LabRedes, Welber, e aos demais amigos e colegas que contribuíram, direta ou indiretamente, para a conclusão deste trabalho, meus sinceros agradecimentos pelo companheirismo e troca de experiências.

Agradeço, ainda, o suporte técnico e computacional do Laboratório de Tecnologias para Tomada de Decisão (LATITUDE) da Universidade de Brasília. Este trabalho foi viabilizado graças ao apoio das seguintes instituições e fomentos: CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico (Outorgas 312180/2019-5 PQ-2 e 465741/2014-2 INCT em Cibersegurança); Advocacia-Geral da União (Outorga AGU 697.935/2019); Procuradoria-Geral da Fazenda Nacional (Outorga PGFN 23106.148934/2019-67); Polícia Federal (Outorga PF 03/2020); Mestrado Profissional em Engenharia Elétrica na área de concentração em Segurança Cibernética – 1ª Turma para Profissionais do Setor de Inteligência (Outorga ABIN 01/2019); Decanatos de Pesquisa e Inovação e de Pós-Graduação da Universidade de Brasília (Outorga 7129 FUB/EMENDA/DPI/COPEI/AMORIS); Projeto SISTER City – Sistemas Inteligentes Seguros e em Tempo Efetivo Real para Cidades Inteligentes (Outorga 625/2022); e Fundação de Apoio à Pesquisa do Distrito Federal (FAP/DF).

RESUMO

A proliferação de *botnets* baseadas na Internet das Coisas (IoT), da qual a família Mirai é o exemplo mais expressivo, exige defesas de borda capazes de conter ataques volumétricos em segundos. As Redes Definidas por Software (SDN, do inglês *Software-Defined Networking*) oferecem essa capacidade ao centralizar o controle da rede. A eficácia dessa reação, porém, pressupõe uma condição raramente examinada pela literatura: a de que as regras emitidas pelo controlador sejam executadas, de forma íntegra e tempestiva, pelo *hardware* que as recebe. Esta dissertação demonstra, em ambiente *Hardware-in-the-Loop* que integra o controlador ONOS a um *switch* comercial, que essa condição falha sob estresse volumétrico. A falha materializa-se nas regras órfãs: regras de bloqueio que o controlador considera removidas, mas que persistem no plano de dados do equipamento e continuam a descartar tráfego legítimo de forma silenciosa. Para restaurar a consistência entre os planos, propõe-se o orquestrador DDSO (*DDoS SDN Orchestrator*), que audita os contadores do *hardware* em malha fechada por meio de um verificador de estado (*State Checker*) e adapta a âncora do bloqueio, escalando do endereço IP para o endereço físico quando há evidência de origem forjada. A validação abrangeu quatro vetores de inundação da família Mirai (SYN, ACK, GRE e UDP), com trinta repetições por condição e intervalos de confiança de 95%. A defesa reduziu o tempo em que o alvo permanece sob ataque (tempo de exposição) entre 91,8% e 96,7%, conforme o vetor. No cenário de origem forjada, em que o bloqueio convencional por endereço IP reduziu a exposição em apenas 1,0%, a escalada para o endereço físico alcançou 96,7%. Conclui-se que a viabilidade operacional da defesa SDN em equipamentos comerciais depende menos da velocidade com que o controlador decide e mais da garantia de que cada decisão se concretize no plano de dados, garantia que os mecanismos propostos sustentaram, nos vetores avaliados, exclusivamente em *software*, sem modificação do equipamento.

Palavras-chave - Redes Definidas por Software (SDN); Segurança IoT; Botnet Mirai; OpenFlow; Persistência de Regras; Hardware-in-the-Loop; Controlador ONOS.

ABSTRACT

The proliferation of Internet of Things (IoT) botnets, of which the Mirai family is the most prominent example, demands edge defenses capable of containing volumetric attacks within seconds. Software-Defined Networking (SDN) offers this capability by centralizing network control. The effectiveness of this reaction, however, rests on a condition rarely examined in the literature: that the rules issued by the controller are executed, fully and in a timely manner, by the hardware that receives them. This dissertation demonstrates, in a Hardware-in-the-Loop environment integrating the ONOS controller with a commercial switch, that this condition fails under volumetric stress. The failure materializes as orphan rules: blocking rules that the controller

considers removed, but which persist in the equipment data plane and silently continue to discard legitimate traffic. To restore consistency between the planes, this work proposes the DDSO (DDoS SDN Orchestrator), which audits the hardware counters in a closed loop through a State Checker and adapts the blocking anchor, escalating from the IP address to the physical address upon evidence of spoofed origin. The validation covered four Mirai-family flooding vectors (SYN, ACK, GRE, and UDP), with thirty repetitions per condition and 95% confidence intervals. The defense reduced the time during which the target remains under attack (exposure time) by 91.8% to 96.7%, depending on the vector. In the spoofed-origin scenario, in which conventional IP-based blocking reduced exposure by only 1.0%, the escalation to the physical address achieved 96.7%. It is concluded that the operational viability of SDN defense on commercial equipment depends less on how fast the controller decides and more on the guarantee that each decision takes effect in the data plane, a guarantee that the proposed mechanisms sustained, across the evaluated vectors, entirely in software, without modifying the equipment.

Keywords - Software-Defined Networking (SDN); IoT Security; Mirai Botnet; OpenFlow; Rule Persistence; Hardware-in-the-Loop; ONOS Controller.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACL	*Access Control List* (Lista de Controle de Acesso)
API	*Application Programming Interface* (Interface de Programação de Aplicações)
ARP	*Address Resolution Protocol* (Protocolo de Resolução de Endereços)
ASIC	*Application-Specific Integrated Circuit* (Circuito Integrado de Aplicação Específica)
BPDU	*Bridge Protocol Data Unit* (Unidade de Dados de Protocolo de Ponte)
C2	*Command and Control* (Comando e Controle)
CAM	*Content-Addressable Memory* (Memória Endereçável por Conteúdo)
CLI	*Command Line Interface* (Interface de Linha de Comando)
CPU	*Central Processing Unit* (Unidade Central de Processamento)
CSV	*Comma-Separated Values* (Valores Separados por Vírgula)
DDoS	*Distributed Denial of Service* (Negação de Serviço Distribuída)
DDSO	*DDoS SDN Orchestrator* (Orquestrador SDN para Mitigação de DDoS)
DNS	*Domain Name System* (Sistema de Nomes de Domínio)
DPI	*Deep Packet Inspection* (Inspeção Profunda de Pacotes)
EDR	*Endpoint Detection and Response* (Detecção e Resposta em *Endpoints*)
EMA	*Exponential Moving Average* (Média Móvel Exponencial)
FDB	*Forwarding Database* (Base de Dados de Encaminhamento)
FPGA	*Field-Programmable Gate Array* (Arranjo de Portas Programável em Campo)
GRE	*Generic Routing Encapsulation* (Encapsulamento de Roteamento Genérico)
gRPC	*gRPC Remote Procedure Calls* (*framework* de chamada de procedimento remoto)
GRU	*Gated Recurrent Unit* (Unidade Recorrente com Comporta)
GVRP	*GARP VLAN Registration Protocol* (Protocolo de Registro de VLAN GARP)
HIDS	*Host-based Intrusion Detection System* (Sistema de Detecção de Intrusão em Hospedeiro)
HIL	*Hardware-in-the-Loop*
HTTP	*Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto)
ICMP	*Internet Control Message Protocol*
IDS	*Intrusion Detection System* (Sistema de Detecção de Intrusão)
IoT	*Internet of Things* (Internet das Coisas)
IP	*Internet Protocol* (Protocolo de Internet)
IPS	*Intrusion Prevention System* (Sistema de Prevenção de Intrusão)
IPv4	*Internet Protocol version 4* (Protocolo de Internet versão 4)
IPv6	*Internet Protocol version 6* (Protocolo de Internet versão 6)
JSON	*JavaScript Object Notation*
KVM	*Kernel-based Virtual Machine* (Máquina Virtual Baseada em *Kernel*)
LACP	*Link Aggregation Control Protocol* (Protocolo de Controle de Agregação de Enlaces)
LAG	*Link Aggregation Group* (Grupo de Agregação de Enlaces)
LLDP	*Link Layer Discovery Protocol* (Protocolo de Descoberta da Camada de Enlace)
LVM	*Logical Volume Manager* (Gerenciador de Volume Lógico)
MAC	*Media Access Control* (Controle de Acesso ao Meio)
ML	*Machine Learning* (Aprendizado de Máquina)
NAT	*Network Address Translation* (Tradução de Endereços de Rede)
NBA	*Network Behavior Analysis* (Análise de Comportamento de Rede)
NBI	*Northbound Interface* (Interface Norte)

NETCONF	*Network Configuration Protocol* (Protocolo de Configuração de Rede)
NFV	*Network Functions Virtualization* (Virtualização das Funções de Rede)
NPU	*Network Processing Unit* (Unidade de Processamento de Rede)
ODL	OpenDaylight (controlador SDN OpenDaylight)
ONIE	*Open Network Install Environment* (Ambiente Aberto de Instalação de Rede)
ONOS	*Open Network Operating System*
OPEX	*Operational Expenditure* (Despesa Operacional)
OS10	Dell SmartFabric OS10 (Sistema operacional de rede da Dell)
OSGi	*Open Services Gateway initiative*
OVS	*Open vSwitch*
PPS	*Packets Per Second* (Pacotes por Segundo)
PVE	*Proxmox Virtual Environment*
QoS	*Quality of Service* (Qualidade de Serviço)
RAM	*Random Access Memory* (Memória de Acesso Aleatório)
REST	*Representational State Transfer* (Transferência de Estado Representacional)
RTT	*Round-Trip Time* (Tempo de Ida e Volta)
SBI	*Southbound Interface* (Interface Sul)
SDN	*Software-Defined Networking* (Redes Definidas por Software)
SSH	*Secure Shell*
STP	*Spanning Tree Protocol* (Protocolo de Árvore de Abrangência)
SVM	*Support Vector Machine* (Máquina de Vetores de Suporte)
TCAM	*Ternary Content-Addressable Memory*
TCP	*Transmission Control Protocol* (Protocolo de Controle de Transmissão)
TLS	*Transport Layer Security* (Segurança da Camada de Transporte)
UDP	*User Datagram Protocol* (Protocolo de Datagrama do Usuário)
VLAN	*Virtual Local Area Network* (Rede Local Virtual)
VM	*Virtual Machine* (Máquina Virtual)
ZFS	*Zettabyte File System*

1. INTRODUÇÃO

As Redes Definidas por Software (SDN, do inglês *Software-Defined Networking*) responderam à rigidez das redes tradicionais ao separar o plano de controle do plano de dados, viabilizando uma gestão centralizada, programável e dinâmica dos recursos [1, 2]. Essa programabilidade converteu a infraestrutura de rede, antes restrita ao encaminhamento estático, em um instrumento ativo de defesa: o controlador, dotado de visão global da topologia, pode reagir a anomalias de tráfego e instalar contramedidas diretamente nos equipamentos de borda [3]. A eficácia dessa reação, entretanto, pressupõe uma condição raramente examinada pela literatura: a de que as decisões emitidas pelo plano de controle sejam refletidas, de forma íntegra e tempestiva, no plano de dados que as executa.

Em paralelo, a expansão da Internet das Coisas (IoT) ampliou a superfície de ataque e facilitou a formação de *botnets*, isto é, redes de dispositivos comprometidos empregados para orquestrar ataques de Negação de Serviço Distribuída (DDoS), de que a família Mirai é o exemplo mais expressivo [4]. Defesas convencionais, baseadas em *appliances* de perímetro estáticos, mostram-se insuficientes diante da volumetria e da agilidade dessas ameaças [5]. A literatura recente aponta a orquestração provida por controladores SDN como resposta eficaz para aplicar contramedidas na borda da rede [6, 7]; contudo, predominam validações em ambientes estritamente simulados, como o Mininet [8].

Essa predominância de validações virtuais constitui uma lacuna relevante, pois ignora as limitações físicas inerentes aos *switches* comerciais, sobretudo as de processamento e de memória de fluxos, com destaque para a memória ternária endereçável por conteúdo (TCAM) [9]. Sob estresse volumétrico, tais limitações comprometem justamente a condição enunciada acima: a remoção lógica de uma regra no controlador deixa de corresponder à sua exclusão efetiva no *hardware*, e o plano de dados passa a abrigar regras órfãs, invisíveis à visão lógica da rede e capazes de descartar tráfego legítimo de forma silenciosa.

Este trabalho avalia empiricamente uma estratégia de defesa de borda orientada por SDN em ambiente *Hardware-in-the-Loop* (HIL), integrando o controlador ONOS (*Open Network Operating System*) a um *switch* comercial. O objetivo é caracterizar os efeitos da assincronia na propagação de regras via protocolo OpenFlow e propor mecanismos de compensação em software, reunidos no orquestrador DDSO (*DDoS SDN Orchestrator*), capazes de preservar a consistência entre o plano de controle e o plano de dados físico durante ataques automatizados.

A estratégia de compensação reúne dois mecanismos complementares, implementados no orquestrador DDSO. O primeiro é o verificador de consistência em malha fechada (*State Checker*), que substitui a confiança no retorno lógico do controlador pela auditoria periódica dos contadores do equipamento. Quando uma regra removida logicamente continua a contabilizar pacotes, o verificador reconhece a inconsistência e reemite a remoção até que o plano de dados reflita o estado

decidido pelo controlador.

O segundo mecanismo é a ancoragem adaptativa da regra de descarte. O bloqueio permanece ancorado no endereço IP de origem enquanto esse identificador se mantém eficaz, por ser o critério mais preciso e de menor efeito colateral. Diante de evidência de falsificação de origem, a âncora escala para o endereço físico do dispositivo, sustentando a mitigação quando o identificador de rede deixa de ser confiável.

A validação experimental submete a arquitetura a quatro vetores volumétricos da família Mirai, as inundações SYN, ACK, GRE e UDP, com trinta repetições por condição e intervalos de confiança de 95%. Os resultados demonstram reduções do tempo em que o alvo permanece sob ataque entre 91,8% e 96,7%, conforme o vetor e a estratégia de ancoragem, e confirmam a eficácia da escalada no cenário de origem forjada, no qual o bloqueio convencional por endereço IP perde o efeito.

1.1 DEFINIÇÃO DO PROBLEMA E QUESTÕES DE PESQUISA

O problema investigado nesta dissertação pode ser enunciado nos seguintes termos: durante a mitigação automatizada de ataques volumétricos, o plano de dados de um *switch* comercial não acompanha, de forma síncrona, as decisões do plano de controle. Dessa assincronia decorrem dois efeitos operacionais adversos. O primeiro é a persistência de regras órfãs no plano de dados físico, entradas que o controlador considera removidas, mas que permanecem ativas no equipamento e continuam a descartar tráfego após o encerramento da ameaça. O segundo é a fragilidade da regra de bloqueio ancorada no endereço IP de origem quando o atacante emprega falsificação dinâmica de endereços, condição em que o identificador de rede deixa de ser confiável e a mitigação convencional perde o efeito. Ambos os efeitos tendem a permanecer ocultos nas validações conduzidas em ambientes emulados, nos quais a tabela de fluxos é uma estrutura de dados em memória, sem os limites e os tempos de processamento do *hardware* real.

A partir desse enunciado, a investigação foi orientada por três questões de pesquisa:

- **QP1:** em que medida a validação em *hardware* comercial revela falhas de sincronização e de persistência de regras que permanecem invisíveis nas validações conduzidas em emuladores?
- **QP2:** é possível restaurar a consistência entre os planos de controle e de dados por meio de mecanismos de compensação implementados exclusivamente em software, sem modificação do equipamento?
- **QP3:** como preservar a eficácia da mitigação quando a falsificação de origem invalida o endereço IP como critério de bloqueio?

A cada questão corresponde uma hipótese, formulada antes da fase experimental e submetida à verificação empírica:

- **H1:** sob estresse volumétrico, a assincronia do protocolo OpenFlow entre a remoção lógica de regras no controlador e a sua efetivação no equipamento comercial produz regras órfãs no plano de dados físico, comportamento não capturado pelas validações conduzidas em emuladores;
- **H2:** a auditoria ativa dos contadores de *hardware*, operando em malha fechada, detecta e elimina essas inconsistências, restaurando a correspondência entre a visão lógica do controlador e o estado real do *switch*;
- **H3:** a ancoragem adaptativa da regra de descarte, com migração do endereço IP para o endereço físico diante de evidência de origem forjada, preserva a eficácia da mitigação em vetores com falsificação de origem.

A verificação dessas hipóteses é objeto do Capítulo 4, que confronta cada uma delas com medições obtidas em repetições controladas de quatro vetores de ataque da família Mirai. Desse percurso resultam as três contribuições centrais do trabalho, a saber: a caracterização empírica do fenômeno das regras órfãs em um *switch* comercial sob ataque volumétrico; o verificador de consistência em malha fechada, que substitui a confiança no retorno lógico do controlador pela auditoria dos contadores do equipamento; e a estratégia de ancoragem adaptativa, que sustenta o bloqueio quando a origem do tráfego é forjada.

1.2 JUSTIFICATIVA

A justificativa para o desenvolvimento deste trabalho apoia-se em razões de ordem técnica, estratégica e acadêmica.

Sob a ótica técnica, a escolha do controlador ONOS justifica-se por sua arquitetura distribuída e voltada a redes de provedores de serviço (*carrier-grade*). Controladores frequentemente adotados em estudos acadêmicos, como POX ou Ryu, apresentam limitações de desempenho e escalabilidade sob estresse volumétrico [10], ao passo que o ONOS oferece alta disponibilidade nativa. Como discutido por Souza e Arima [11], a contenção de *botnets* em ambientes críticos exige uma gestão de estado e uma agilidade na manipulação de fluxos que controladores mais simples, ou equipamentos tradicionais rígidos, não sustentam.

Do ponto de vista estratégico e econômico, a mitigação eficaz na borda da rede (*edge defense*) é condição necessária para preservar a disponibilidade de serviços essenciais. A integração de segurança intrínseca à infraestrutura (*security by design*) via SDN reduz a dependência de *appliances* proprietários inflexíveis e contém custos operacionais (OPEX). Neutralizar ataques volumétricos o mais próximo possível de sua origem evita a saturação dos enlaces centrais e protege o núcleo da rede contra o esgotamento de recursos [12].

No âmbito acadêmico, esta pesquisa aborda uma lacuna pouco explorada na literatura de segurança em SDN: a transição de ambientes puramente simulados para infraestruturas físicas.

Embora autores como Aslam *et al.* [6] destaquem a necessidade de módulos de segurança adaptativos, predomina o uso de emuladores como o Mininet [8]. Essa abordagem mascara falhas operacionais inerentes a equipamentos físicos, como a assincronia na atualização de tabelas TCAM. A contribuição desta dissertação reside, portanto, na validação em ambiente *Hardware-in-the-Loop*, que evidencia as limitações da integração de *switches* comerciais com o controlador ONOS e fornece mecanismos empíricos para garantir a consistência das regras de segurança.

1.3 OBJETIVOS

Esta seção apresenta os objetivos que norteiam o desenvolvimento deste trabalho.

1.3.1 Objetivo Geral

O objetivo principal desta dissertação é propor, implementar e avaliar empiricamente uma arquitetura de defesa de borda baseada em Redes Definidas por Software (SDN), integrando o controlador ONOS a um *switch* comercial em um ambiente *Hardware-in-the-Loop*, com vistas à mitigação automatizada de ataques volumétricos da família Mirai e à resolução de falhas de sincronização e persistência de regras no plano de dados físico.

1.3.2 Objetivos Específicos

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

- **Investigar o estado da arte:** realizar o levantamento bibliográfico sobre as vulnerabilidades de redes IoT frente a ataques volumétricos e identificar as limitações operacionais das propostas de defesa SDN restritas a simuladores de rede;
- **Estruturar a topologia de experimentação:** projetar e implementar um ambiente de testes realista em modelo *Hardware-in-the-Loop*, operacionalizando a comunicação via protocolo OpenFlow entre o controlador ONOS e um *switch* físico comercial (Dell S3048-ON);
- **Reproduzir cenários de ameaça:** empregar o código da botnet Mirai em ambiente controlado e isolado para gerar tráfego de ataque com perfis e vetores autênticos, submetendo a infraestrutura a condições de estresse volumétrico;
- **Caracterizar o comportamento do *hardware*:** avaliar as limitações do plano de dados físico durante a mitigação, identificando inconsistências operacionais, como a assincronia do protocolo e a geração de regras órfãs na memória do *switch*;
- **Desenvolver mecanismos de compensação:** criar e integrar rotinas de verificação de estado no aplicativo de segurança do ONOS, capazes de detectar e corrigir dinamicamente as falhas de sincronização entre o controlador e o *hardware*;

- **Validar o desempenho da solução:** medir e comparar a eficácia da arquitetura por meio do tempo de exposição do alvo, da latência de mitigação, da latência de escalada da ancoragem e da persistência de regras no plano de dados, demonstrando a viabilidade da defesa em equipamentos comerciais.

1.4 METODOLOGIA DE PESQUISA

A presente dissertação caracteriza-se como uma pesquisa aplicada, de natureza exploratória e descritiva, conduzida sob uma abordagem mista (quantitativa e qualitativa). Conforme Gil [13], pesquisas aplicadas buscam gerar conhecimentos orientados à solução de problemas concretos. A experimentação prática justifica-se pela necessidade de transcender as simulações puramente virtuais, avaliando empiricamente o comportamento de equipamentos comerciais físicos sob estresse volumétrico.

O percurso metodológico foi estruturado em seis etapas principais, descritas a seguir quanto aos procedimentos e às ferramentas empregadas:

- **Revisão de literatura:** levantamento bibliográfico estruturado sobre segurança em redes SDN, mitigação de ataques DDoS focados em IoT (família Mirai) e a arquitetura do controlador ONOS, conduzido segundo as orientações de Marconi e Lakatos [14];
- **Modelagem do problema de pesquisa:** caracterização das vulnerabilidades da integração entre o plano de controle centralizado e o plano de dados físico, com a formulação das questões de pesquisa e das hipóteses apresentadas na Seção 1.1, relativas à assincronia do protocolo OpenFlow e às restrições de memória TCAM em *switches* comerciais durante a persistência de regras de bloqueio;
- **Estruturação do ambiente HIL:** integração do controlador ONOS a um *switch* físico comercial (Dell S3048-ON), reproduzindo as condições de uma infraestrutura de borda;
- **Reprodução de ameaças volumétricas:** execução controlada da *botnet* Mirai em ambiente isolado de laboratório, gerando tráfego de ataque com vetores autênticos mesclados a fluxos legítimos, de modo a submeter a tabela de fluxos do equipamento a cenários de estresse e forçar a ocorrência de inconsistências de estado e regras órfãs;
- **Desenvolvimento e aplicação de mecanismos de compensação:** projeto e implementação de rotinas de verificação de estado no módulo de segurança do ONOS (orquestrador DDSO), atuando como contramedidas às falhas do *hardware* para garantir a sincronização entre os planos de controle e de dados;
- **Análise crítica e avaliação de desempenho:** coleta de métricas quantitativas (tempo de exposição do alvo; latência de mitigação; latência de escalada da ancoragem; e persistência de regras no plano de dados), comparadas entre cenários com e sem os mecanismos

de compensação ativos, de modo a confrontar as hipóteses H1, H2 e H3 com os dados experimentais.

Os dados empíricos foram extraídos diretamente dos *logs* do controlador e das tabelas de fluxo (*flow tables*) do *switch* físico. Embora ferramentas de emulação como o Mininet sejam amplamente utilizadas na literatura para validar lógicas de controle [15], elas mascaram o comportamento assíncrono do *hardware* e o tempo real de processamento interno dos equipamentos, como os gargalos na CPU do *switch* e na TCAM. Por isso, a coleta concentrou-se no ambiente físico, utilizando campos essenciais do cabeçalho OpenFlow para aplicar bloqueios precisos.

Na automação de segurança de redes, a aplicação intermitente de bloqueios, ou a falha na remoção de regras (regras órfãs), pode gerar indisponibilidade permanente de serviços legítimos. O sucesso desta metodologia dependeu, portanto, de uma abordagem, no controlador, que compensasse a incapacidade do *hardware* de manter o sincronismo sob ataques contínuos.

A metodologia adotada consolida fundamentos teóricos de Redes Definidas por Software com uma validação laboratorial controlada. O caráter aplicado da pesquisa demonstra, como prova de conceito, a viabilidade operacional de uma defesa de borda em equipamentos comerciais frente a ameaças modernas de IoT.

1.5 PUBLICAÇÕES

O desenvolvimento desta pesquisa está associado a dois trabalhos científicos submetidos para publicação, relacionados a seguir [16, 17]. O primeiro reúne a validação empírica em rede física e o segundo, a fundamentação conceitual em arquiteturas de próxima geração.

- **Trabalho principal (submetido)** [16]:
  - **Título:** "Defesa de Borda Orientada a SDN: Desafios de Sincronização e Persistência de Regras na Mitigação de Ameaças *Mirai-like*".
  - **Autores:** Clayton Menezes Silva, Victor Lima dos Santos, Georges Daniel Amvame Nze, João José Costa Gondim, Francisco Lopes de Caldas Filho e Fábio Lúcio Lopes de Mendonça.
  - **Destino:** submetido à *21ª Conferência Ibérica de Sistemas e Tecnologias de Informação* (CISTI 2026).
  - **Contribuição:** apresenta a validação empírica da proposta em abordagem *Hardware-in-the-Loop*, integrando o controlador ONOS a um *switch* comercial Dell S3048-ON. Identifica e soluciona problemas de *hardware* invisíveis em simulações, como as regras órfãs que persistem na memória TCAM e a instabilidade na identificação de fluxos, demonstrando sua neutralização por meio de um mecanismo de verificação de consistência em malha fechada.

- **Trabalho correlato (submetido)** [17]:
  - **Título:** “Proposal for a 6G-Integrated SDN Security Architecture for Adaptive Mitigation of Mirai Attacks in IoT Ecosystems”.
  - **Autores:** Clayton Menezes Silva, João José Costa Gondim, Jorge Osvaldo Alves de Lima Torres, Georges Daniel Amvame Nze, Robson de Oliveira Albuquerque e Fábio Lúcio Lopes de Mendonça.
  - **Destino:** submetido ao periódico *Engineering Proceedings* (MDPI).
  - **Contribuição:** estabelece, em nível conceitual, as bases de uma arquitetura de defesa de borda nativa para IA em redes 6G, introduzindo o modelo de controle em malha fechada para a consistência de estado. A redução de latência ali estimada (cerca de 35%) refere-se a esse modelo conceitual e não se confunde com as medições empíricas obtidas nesta dissertação.

1.6 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada em cinco capítulos. Além desta introdução, os capítulos seguintes conduzem o leitor da fundamentação teórica à análise dos resultados empíricos obtidos em *hardware* real, conforme descrito a seguir:

- **Capítulo 2, Fundamentação Teórica:** aborda os conceitos fundamentais sobre Redes Definidas por Software (SDN), a arquitetura *carrier-grade* do controlador ONOS e o protocolo OpenFlow. Discute o ecossistema de ameaças IoT, com foco na *botnet* Mirai, e realiza uma revisão do estado da arte, evidenciando a lacuna de validações em equipamentos físicos comerciais.
- **Capítulo 3, Proposta de Arquitetura e Ambiente Experimental:** detalha a concepção e a implementação da topologia em modelo *Hardware-in-the-Loop* (HIL). Descreve a integração entre o controlador ONOS e o *switch* físico (Dell S3048-ON), as ferramentas de geração de ataques volumétricos e a lógica do mecanismo de verificação de estado desenvolvido para compensar a assincronia do *hardware*.
- **Capítulo 4, Análise e Resultados:** apresenta a avaliação empírica da arquitetura sob quatro vetores volumétricos da família Mirai: a inundação SYN de origem fixa, a inundação ACK com falsificação de origem, a inundação GRE com origem real distribuída e a inundação UDP de origem fixa. Caracteriza o fenômeno das regras órfãs no plano de dados e a sua reconciliação pelo verificador de consistência em malha fechada, demonstra a ancoragem adaptativa que escala o bloqueio do endereço IP para o endereço físico diante de origem forjada e identifica, no cenário distribuído, a contenção incremental por origem, em que a

exposição do alvo é governada pela última origem bloqueada. Os resultados são sustentados por intervalos de confiança de 95%.

- **Capítulo 5, Conclusão:** sintetiza as principais contribuições operacionais e acadêmicas da pesquisa, retoma os objetivos alcançados, discute as limitações inerentes aos testes físicos e sugere direcionamentos para trabalhos futuros focados na resiliência de redes SDN.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a base conceitual e tecnológica necessária para a compreensão da arquitetura de defesa proposta e dos fenômenos observados durante a fase experimental desta dissertação. Além de descrever as ferramentas utilizadas, a revisão fundamenta as escolhas de engenharia adotadas para contornar as limitações de infraestruturas físicas sob ataques volumétricos.

O capítulo está organizado em quatro eixos. As Seções 2.1 e 2.2 tratam dos princípios do paradigma de Redes Definidas por Software (SDN), do funcionamento do protocolo OpenFlow e da arquitetura distribuída do controlador ONOS (*Open Network Operating System*), com atenção aos gargalos de comunicação decorrentes da separação entre os planos de controle e de dados. A Seção 2.3 examina o ecossistema de ameaças voltadas à Internet das Coisas (IoT), em particular a mecânica de proliferação e exaustão de recursos da família de *botnets* Mirai, que fundamenta o modelo de ameaça deste trabalho. A Seção 2.4 discute a defesa de borda (*edge defense*) e as limitações de processamento em *switches* comerciais. Por fim, a Seção 2.5 revisa os trabalhos relacionados e delimita a lacuna que esta dissertação endereça.

2.1 REDES DEFINIDAS POR SOFTWARE (SDN) E O PROTOCOLO OPENFLOW

A arquitetura tradicional de redes de computadores, historicamente caracterizada pelo acoplamento rígido entre as lógicas de encaminhamento e de controle dentro do mesmo equipamento, apresenta limitações severas frente às demandas de escalabilidade e segurança contemporâneas. As Redes Definidas por Software (SDN, do inglês *Software-Defined Networking*) respondem a essas limitações ao viabilizar uma gestão de infraestrutura ágil e programável [1]. O princípio basilar da arquitetura SDN consiste em desacoplar o plano de controle (responsável pela tomada de decisão e roteamento) do plano de dados (responsável pelo encaminhamento de pacotes em *hardware*).

Essa separação lógica confere ao controlador centralizado uma visão global da topologia: a decisão de encaminhamento de cada pacote passa a depender do estado global da rede e das políticas de segurança ativas no momento, e não apenas de tabelas locais estáticas.

Para que o controlador orquestre o plano de dados, faz-se necessária uma interface de comunicação padronizada, conhecida como *Southbound Interface* (SBI). O protocolo OpenFlow consolidou-se como o padrão *de fato* para essa comunicação [2, 18]. Por meio de mensagens OpenFlow (especificamente as instruções `OFPT_FLOW_MOD`), o controlador insere, modifica ou remove regras diretamente nas tabelas de fluxo (*Flow Tables*) dos comutadores, ditando como cada pacote deve ser tratado assim que ingressa na porta do equipamento.

2.1.1 A Estrutura da Tabela de Fluxos (*Flow Table*)

No OpenFlow, o roteamento clássico baseado apenas em endereço IP de destino ou endereço MAC é substituído pelo conceito de fluxo. A partir da versão 1.3 do protocolo (utilizada nesta pesquisa), permite-se o encadeamento de múltiplas tabelas (*Pipeline Processing*). Cada entrada em uma tabela de fluxos é uma estrutura complexa, cujos componentes são cruciais para o funcionamento de heurísticas de defesa, conforme detalhado na Tabela 2.1.

Tabela 2.1: Componentes estruturais de uma entrada de fluxo (*Flow Entry*) no protocolo OpenFlow.

Campo	**Descrição Técnica**
Match Fields	Tuplas de correspondência que definem a assinatura do pacote (ex: `ETH_SRC`, `IPV4_DST`, `TCP_DST`). Suporta máscaras de bits para sub-redes.
Priority	Valor numérico para resolução de ambiguidades entre regras coincidentes. Garante a precedência das regras de bloqueio sobre o encaminhamento normal.
Counters	Registradores de estatísticas atualizados em *hardware* para cada pacote processado. Fonte primária das métricas de detecção de anomalias.
Instructions	Ações a serem executadas caso haja correspondência (ex: *Forward, Drop, Modify Header*).
Timeouts	Contadores regressivos (*Hard* e *Idle*) que definem o tempo de vida da regra antes de ser expurgada automaticamente.

Fonte: Adaptado de [18].

Apesar da flexibilidade teórica proporcionada pelo SDN e pelo protocolo OpenFlow, a execução física dessas políticas de controle esbarra em restrições intrínsecas ao *hardware*. Em avaliações puramente virtuais, as regras são processadas na memória RAM de hospedeiros emulados, refletindo um estado lógico quase instantâneo. Contudo, em cenários reais, os *switches* comerciais de borda (*Top-of-Rack*) dependem de *Application-Specific Integrated Circuits* (ASICs) para garantir o encaminhamento em velocidade de cabo (*line-rate*) [9].

Para que as regras OpenFlow sejam aplicadas sem degradar o desempenho, elas precisam ser instaladas na memória TCAM (*Ternary Content-Addressable Memory*) do comutador. Diferente da memória RAM, a TCAM possui um tamanho estritamente limitado e apresenta uma latência inerente para a atualização e gravação de novas entradas.

Essa latência de gravação tem consequência direta sob estresse. Durante a mitigação reativa de um ataque volumétrico, o comando emitido pelo controlador e a sua gravação efetiva na TCAM deixam de ser simultâneos. Essa assincronia gera inconsistências de estado entre os planos e compromete a confiabilidade da arquitetura SDN em produção.

2.2 ARQUITETURA DO CONTROLADOR ONOS

A seleção do controlador SDN é uma etapa crítica na modelagem de defesas de borda, pois este elemento centraliza toda a inteligência e o gerenciamento de estado da infraestrutura. Enquanto ferramentas acadêmicas clássicas, como POX ou Ryu, oferecem facilidade de prototipagem, elas frequentemente apresentam limitações de escalabilidade e resiliência quando submetidas a rajadas de tráfego volumétrico [10, 19]. A Tabela 2.2 compara as principais soluções disponíveis e fundamenta a escolha tecnológica desta pesquisa.

Tabela 2.2: Análise comparativa entre controladores SDN de uso acadêmico e comercial.

Critério	**ONOS**	**OpenDaylight (ODL)**	**Ryu**
Linguagem	Java (OSGi)	Java (Karaf)	Python
Foco Principal	Provedores e Telcos	Ambientes *Enterprise*	Prototipagem Acadêmica
Arquitetura	Modular e Distribuída	Baseada em Modelos	Monolítica/Modular
Consistência	Atomix (Algoritmo Raft)	Raft / Akka	Centralizada
Desempenho	Alta Escalabilidade	Alta Flexibilidade	Moderado (Gargalo de I/O)

Fonte: Adaptado de [20, 10].

A adoção do *Open Network Operating System* (ONOS), portanto, baseia-se na sua natureza *carrier-grade*, projetada desde a concepção para garantir alta disponibilidade [20]. O diferencial técnico do ONOS reside na sua arquitetura modular, que segrega as funções em quatro camadas: a camada de Aplicações SDN; a interface de comunicação superior (*Northbound API*); o Núcleo Distribuído (*Distributed Core*), suportado pelo *framework* Atomix [21]; e a interface de comunicação com os equipamentos (*Southbound*). Essa separação permite que aplicações externas de segurança, como o motor heurístico de detecção de anomalias desenvolvido neste trabalho, atuem no topo da pilha, enviando diretrizes de bloqueio em formato JSON via chamadas RESTful, sem precisar interagir diretamente com as complexidades do protocolo OpenFlow.

Entretanto, a utilização dessa arquitetura em camadas introduz um desafio direto de sincronização. Conforme será detalhado na topologia deste estudo, a aplicação reativa de políticas precisa atravessar a interface *Northbound*, ser processada pelo Núcleo Distribuído e, finalmente, ser traduzida pela interface *Southbound* para o *chipset* do *switch* físico. Para reduzir o tempo de reação e atingir uma latência de mitigação eficaz, o mecanismo de defesa desenvolvido precisou operar diretamente sobre a API de Fluxos (*Flow Rule API*), contornando o *Application Intent Framework* nativo da plataforma.

Embora essa decisão arquitetural de desvio (*bypass*) assegure a rapidez necessária para conter ataques em sua rampa inicial, ela abdica das ferramentas nativas de reconciliação do controlador. Ao preterir o *Intent Framework* em favor da velocidade de injeção direta, o sistema transfere a responsabilidade do gerenciamento de estado para a aplicação externa. Essa escolha técnica abre margem para falhas silenciosas de persistência no plano de dados e exige a implementação de mecanismos robustos de auditoria em malha fechada.

A Figura 2.1 ilustra essa organização. As quatro camadas de *software* do controlador, a interface *Southbound*, o Núcleo Distribuído sustentado pelo Atomix, a *Northbound API* e a camada de Aplicações SDN, operam sobre o plano de dados físico, o equipamento controlado, com fluxo de controle bidirecional entre as camadas. O módulo DDSO, destacado na camada de aplicações, ocupa a posição a partir da qual as diretrizes de bloqueio desta pesquisa atravessam a pilha até o *hardware*.

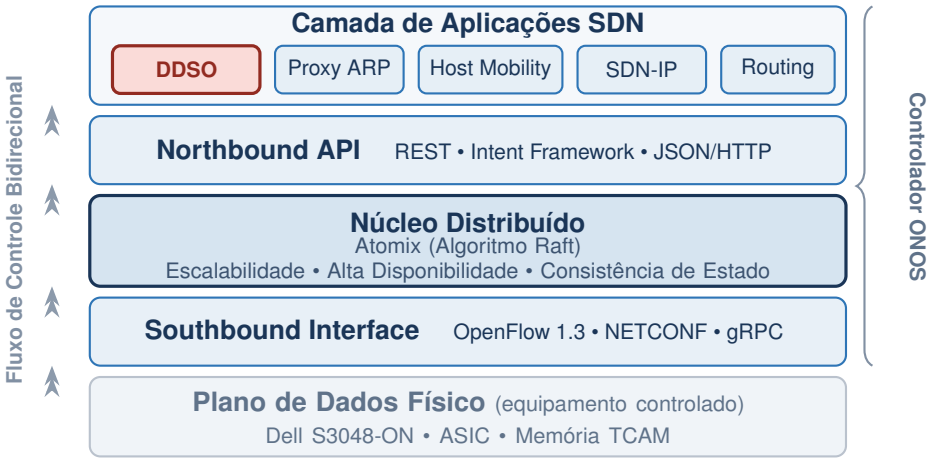

Figura 2.1: Arquitetura modular do controlador ONOS e a sua relação com o plano de dados físico.

2.3 O ECOSSISTEMA DE AMEAÇAS IOT E A BOTNET MIRAI

Uma *botnet* é uma rede de dispositivos comprometidos, denominados *bots* ou nós zumbis, que executam, à revelia de seus proprietários, ordens emitidas de forma remota por um servidor de comando e controle (C2). O controle centralizado de um grande número de dispositivos permite ao operador coordenar ações maliciosas em escala, entre as quais o envio massivo de tráfego contra um alvo comum, que caracteriza os ataques de Negação de Serviço Distribuída (DDoS) [4, 22]. No ecossistema da Internet das Coisas, a abundância de dispositivos expostos e mal protegidos tornou essas redes maliciosas particularmente numerosas e agressivas, contexto em que a família Mirai se consolidou como o exemplo mais expressivo, detalhado nas subseções seguintes.

A rápida proliferação da Internet das Coisas (IoT) redefiniu o perímetro de segurança das redes modernas. Dispositivos embarcados, sensores industriais e câmeras IP introduzem um ecossistema heterogêneo, frequentemente caracterizado por *firmwares* obsoletos e pela completa ausência de práticas de reforço de segurança (*hardening*). Essa insegurança estrutural transforma tais equipamentos em alvos preferenciais e vetores primários para a formação de redes maliciosas em larga escala [22, 23].

As soluções tradicionais de segurança mostram-se ineficazes diante desse cenário descentralizado. Os *firewalls* de borda convencionais operam com regras estáticas e são invariavelmente cegos ao tráfego lateral interno, enquanto sistemas de detecção e resposta em *endpoints* (EDR)

tornam-se inaplicáveis devido à impossibilidade técnica de instalar agentes de *software* em sistemas operacionais IoT que são proprietários e fechados.

Foi essa lacuna estrutural que a *botnet* Mirai explorou [4]. Ao infectar centenas de milhares de nós vulneráveis, a Mirai orquestrou ataques de Negação de Serviço Distribuída (DDoS) cujos picos documentados atingiram a ordem de centenas de gigabits por segundo.

2.3.1 Anatomia e *Kill Chain* da Infecção

A mecânica de propagação e exaustão da Mirai opera em um ciclo de vida automatizado e agressivo, estruturado nas seguintes fases [4]:

1. **Fase de Varredura (*Scanning*):** Dispositivos já infectados geram pacotes `TCP SYN` pseudo-aleatórios direcionados a endereços IPv4 públicos globais, buscando especificamente as portas 23 (Telnet) e 2323 abertas.
2. **Fase de Intrusão (*Brute-Force*):** Ao identificar uma porta aberta, o *malware* tenta obter acesso via força bruta utilizando um dicionário *hardcoded* de 62 combinações de usuários e senhas padrão de fábrica (como `root:root`, `admin:admin`, `support:support`).
3. **Fase de Infecção (*Payload Loader*):** Obtido o acesso, a *botnet* identifica a arquitetura do processador da vítima (ARM, MIPS, x86) via comandos de *shell*, faz o *download* do binário correspondente à sua arquitetura, executa-o diretamente na memória RAM (para dificultar a análise forense no disco) e apaga seus rastros.
4. **Fase de Orquestração (*Command and Control*):** O novo dispositivo "zumbi"conecta-se ao servidor C2 (*Command and Control*) e aguarda diretrizes. Quando o atacante define um alvo, o C2 envia o comando contendo o endereço IP da vítima, a duração do ataque e o vetor (tipo de inundação) a ser utilizado.

A Figura 2.2 sintetiza o panorama da pesquisa: o ciclo de ataque *Mirai-like*, da infecção dos dispositivos IoT à orquestração do DDoS volumétrico, e a defesa de borda orientada por SDN proposta sobre ambiente *Hardware-in-the-Loop*. A lacuna endereçada situa-se na transição do ataque para a defesa, com a validação em *hardware* físico, ausente nos estudos restritos à emulação.

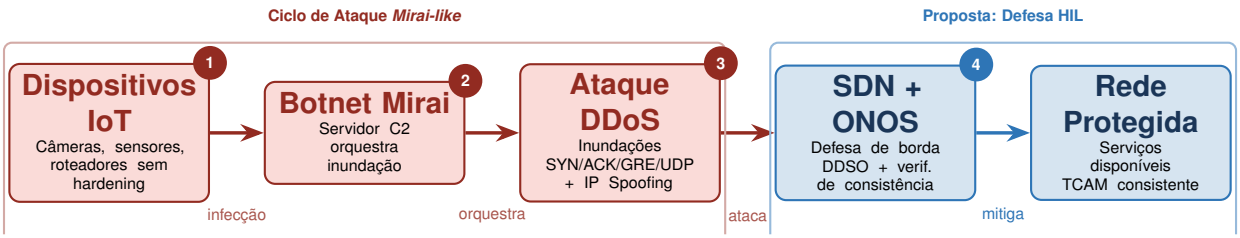

Figura 2.2: Panorama da pesquisa: ciclo de ataque *Mirai-like* e defesa de borda proposta em ambiente *Hardware-in-the-Loop*.

No escopo específico desta pesquisa, a emulação da ameaça concentra-se na assinatura de exaustão volumétrica, materializada nos quatro vetores avaliados no Capítulo 4: as inundações SYN, ACK, GRE e UDP. Diferentemente de ataques na camada de aplicação, esses vetores visam esgotar a capacidade de processamento dos equipamentos de rede, saturando tabelas de estado e enlaces antes mesmo que o tráfego atinja os servidores finais. É essa agressividade volumétrica que a defesa de borda precisa conter antes da saturação da infraestrutura intermediária.

2.4 DEFESA DE BORDA (*EDGE DEFENSE*) EM SDN

A mitigação de ameaças volumétricas modernas exige uma intervenção o mais próxima possível da origem do tráfego anômalo. No contexto de redes corporativas e infraestruturas críticas, a estratégia de defesa de borda (*edge defense*) visa conter os vetores de ataque logo no ponto de ingresso da topologia. Essa abordagem impede a saturação dos enlaces do núcleo (*core*) da rede e preserva integralmente a capacidade de processamento e a disponibilidade dos servidores finais.

A aplicação do paradigma SDN para orquestrar essa defesa introduz a capacidade de proteger a rede sem a necessidade de agentes (*agentless*) [3]. Como os dispositivos IoT sequestrados por *botnets* possuem *firmwares* proprietários que impossibilitam a instalação de ferramentas locais de segurança, a transferência da inteligência de detecção para a rede torna-se o único caminho viável. Nesse modelo, o controlador concentra a função de decisão, enquanto os *switches* físicos de acesso executam o descarte em tempo real.

Na prática, a defesa orquestrada consolida métricas de telemetria da borda (como o volume de pacotes por segundo) e, ao identificar um desvio comportamental característico de uma inundação volumétrica, o módulo de segurança injeta reativamente uma regra de bloqueio (instrução *Drop*) na primeira tabela de fluxos do equipamento, descartando o tráfego no ASIC, em velocidade de cabo [24, 25].

Contudo, para que a defesa de borda baseada em SDN seja verdadeiramente resiliente em ambientes de produção, ela não pode ser concebida sob a premissa de um sincronismo ideal. O plano de controle deve estar preparado para lidar com a latência imposta pelos barramentos físicos e pela alocação na memória TCAM. Garantir que as regras de mitigação sejam aplicadas com rapidez e, de igual modo, expurgadas com precisão após a normalização do tráfego é o elemento crítico que previne o esgotamento do comutador e assegura a viabilidade operacional do SDN em equipamentos comerciais.

2.5 TRABALHOS RELACIONADOS

Esta seção apresenta a revisão dos trabalhos relacionados, posicionando a presente pesquisa em relação à literatura existente em segurança de redes SDN e na mitigação de ataques DDoS em

ambientes IoT. A revisão está organizada em quatro eixos temáticos: a caracterização da ameaça representada pela *botnet* Mirai; as propostas de contenção de ataques DDoS baseadas em SDN; as abordagens fundamentadas em aprendizado de máquina; e os estudos sobre as limitações de desempenho do *hardware* e da consistência de estado em controladores SDN. Ao final, uma síntese comparativa evidencia as lacunas que motivaram a abordagem adotada neste trabalho.

2.5.1 A botnet Mirai e a superfície de ataque em IoT

O trabalho de Antonakakis *et al.* [4] constitui referência fundamental para o estudo de *botnets* IoT de larga escala. Publicada no USENIX Security de 2017, a análise reconstruiu, a partir de dados reais de propagação, a mecânica interna de uma *botnet* que comprometeu centenas de milhares de dispositivos. Além da escala do fenômeno, o estudo destaca a baixa complexidade das credenciais exploradas: o *malware* utilizava um dicionário de 62 pares de usuário e senha, em sua maioria variações de combinações padrão de fábrica. Essa característica possui implicações diretas para a heurística implementada no módulo DDSO, uma vez que as fases de varredura e intrusão da Mirai geram um padrão de tráfego específico e identificável sem a necessidade de inspeção profunda de pacotes (DPI).

Kolias *et al.* [22] ampliaram esse diagnóstico ao contextualizar a Mirai no panorama mais amplo das ameaças DDoS em infraestruturas IoT. A contribuição desse trabalho é de natureza analítica, argumentando que o problema é predominantemente estrutural, e não tecnológico: a priorização de custo e de velocidade de comercialização em detrimento da segurança resulta em um ecossistema de dispositivos vulneráveis desde a concepção (*by design*). Esse argumento fundamenta a decisão, adotada nesta pesquisa, de não depender de agentes instalados nos dispositivos para a realização da defesa, abordagem inviável em redes compostas por câmeras IP e roteadores embarcados.

A terceira referência central desse eixo é o trabalho de Meidan *et al.* [23] sobre o N-BaIoT. Os autores demonstraram que dispositivos IoT infectados exibem comportamentos de tráfego suficientemente estáveis para permitir a detecção por anomalia, dispensando a DPI. O resultado, contudo, apresenta uma limitação relevante: o N-BaIoT foi concebido para operar individualmente, por dispositivo. No cenário desta dissertação, em que os nós atacantes são máquinas virtuais conectadas ao *switch* físico Dell S3048-ON e os dispositivos IoT comprometidos são, por natureza, fechados e inacessíveis à instrumentação local, a inteligência de detecção precisa residir na rede, e não nos *endpoints*, o que justifica a adoção da abordagem SDN.

2.5.2 Propostas de contenção de DDoS via SDN

Braga *et al.* [24] demonstraram, em 2010, que as métricas de fluxo do protocolo OpenFlow são suficientes para detectar inundações volumétricas sem a inspeção do conteúdo dos pacotes. A abordagem de Análise de Comportamento de Rede (NBA, *Network Behavior Analysis*), formalizada com o controlador NOX, influenciou diretamente a heurística implementada no DDSO. A principal limitação do trabalho reside no fato de a validação ter sido conduzida exclusivamente

em emulação, condição que impede a observação dos gargalos que emergem ao aplicar a mesma lógica em *hardware* físico sob estresse real, situação não enfrentada pelo NOX emulado.

Giotis *et al.* [25] avançaram nessa direção ao combinar o OpenFlow com a exportação de amostras via sFlow, propondo um mecanismo de detecção potencialmente mais escalável. Embora o raciocínio sobre escalabilidade seja consistente em termos teóricos, a validação restringiu-se ao emulador Mininet. Como consequência, o consumo de memória TCAM pelas regras de mitigação e o comportamento do sistema diante do esgotamento desse recurso não foram avaliados.

Scott-Hayward *et al.* [3] produziram um *survey* abrangente sobre os mecanismos de segurança em SDN, útil como mapeamento geral do campo. O trabalho cataloga os vetores de ataque contra a arquitetura SDN, incluindo os ataques de exaustão do canal de controle por meio de tempestades de `PACKET_IN`, fenômeno central nesta dissertação. Por se tratar de um *survey*, entretanto, o estudo não fornece evidência empírica sobre a manifestação desses fenômenos em *hardware* real, evidência que os experimentos conduzidos nesta pesquisa buscam produzir.

No contexto nacional, Souza e Arima [11] investigaram a aplicação do ONOS para a detecção de *malware* em ambientes IoT. A escolha do controlador é convergente com a desta dissertação, o que facilita a comparação direta. O trabalho, no entanto, permanece restrito a um ambiente virtualizado, não abordando o comportamento do mecanismo quando submetido a um *switch* físico com TCAM limitada e ao volume de mensagens `PACKET_IN` característico desse cenário.

Uma linha mais recente de pesquisa deslocou a mitigação para o próprio plano de dados programável. Sobre comutadores P4 de alto desempenho, o Poseidon [26] e o Jaqen [27] detectam e contêm inundações volumétricas no silício, sem recorrer a um controlador externo. O Ripple [28] estendeu a ideia a uma defesa descentralizada contra adversários adaptativos, e o Euclid [29] a implementou de forma totalmente *in-network*. Duas extensões recentes ampliaram esse alcance: o Mew [30] trata inundações de enlace dinâmicas por meio de migração de estado, e o trabalho de Yan *et al.* [31] embarca modelos de aprendizado de máquina em código P4 para classificar tipos distintos de DDoS sob a ótica de redes 6G.

Esses trabalhos definem o estado da arte em desempenho, mas partem de uma mesma premissa: um plano de dados reprogramável, como o das arquiteturas Tofino. Essa premissa não se aplica ao comutador OpenFlow de função fixa avaliado nesta dissertação, cujo agente expõe um *pipeline* de tabela única. A defesa, nesse cenário, precisa ser orquestrada externamente, e não recodificada no silício.

2.5.3 Abordagens baseadas em aprendizado de máquina

A literatura recente em SDN e segurança IoT apresenta uma tendência crescente de tratar o problema de detecção por meio de modelos de aprendizado de máquina cada vez mais sofisticados. Shinan *et al.* [8] mapearam esse cenário em uma revisão sistemática publicada em 2021, documentando que abordagens supervisionadas, como *Random Forest*, SVM e redes neurais, alcançam elevadas taxas de acurácia em *benchmarks* padronizados. Como reconhecido pelos

próprios autores, contudo, tais *benchmarks* foram produzidos majoritariamente em ambientes emulados, de modo que a latência de inferência, fator potencialmente decisivo em dispositivos de borda, não é considerada.

Romani *et al.* [32] e Mbongo *et al.* [33] estendem essa tendência com arquiteturas mais elaboradas, como aprendizado federado e redes Conv1D-GRU com mecanismo de atenção, reportando índices de precisão superiores a 99% em alguns casos. Para o cenário desta dissertação, entretanto, permanece em aberto a questão da viabilidade temporal: esses trabalhos não reportam a latência de inferência dos modelos, fator que, em um ciclo de reação como o avaliado nesta pesquisa (consulta aos contadores a cada 1 s e latência de mitigação média de 0,71 s), somaria atraso diretamente ao tempo de exposição do alvo. A opção por heurísticas determinísticas leves no DDSO não decorre do desconhecimento das técnicas mais avançadas, mas de uma decisão metodológica de isolar o problema efetivamente investigado: não a acurácia da detecção, mas a consistência da execução no plano de dados físico.

Kalimuthu e Velumani [34] exploram essa fronteira em um contexto distinto, propondo um sistema de detecção de intrusão para ambientes IoT baseado em otimização adaptativa com aprendizado profundo. O trabalho é representativo de uma vertente em rápido crescimento, porém igualmente distante da validação em *hardware* SDN real. Aslam *et al.* [6] constituem um caso particular, ao empregarem o próprio ONOS como plataforma de defesa reativa, o que aproxima o estudo do contexto desta pesquisa. Os autores analisam múltiplos *datasets* de ataque e avaliam a resposta do controlador; a questão da persistência indevida de regras no plano de dados após a mitigação, fenômeno central nesta dissertação, não é, porém, abordada.

2.5.4 Limitações de hardware e consistência de estado em SDN

Tootoonchian *et al.* [10] foram, em 2012, pioneiros na avaliação sistemática do desempenho de controladores SDN sob carga. O resultado central indica que a capacidade de processamento de mensagens `PACKET_IN` constitui o gargalo determinante do limite prático da arquitetura, sobrepondo-se à lógica de controle e à largura de banda da rede no tratamento de eventos de *table-miss*. Esse gargalo no processamento de `PACKET_IN` é um dos fatores que tornam o comportamento de *switches* físicos sob estresse distinto do observado em emuladores, distinção que motiva a investigação empírica conduzida nesta dissertação.

Wijeratne *et al.* [9] aprofundam essa análise ao investigar as restrições físicas de memória e latência em *hardware* SDN real, especificamente em FPGAs. Trata-se do trabalho metodologicamente mais próximo desta dissertação, por utilizar *hardware* físico e mensurar comportamentos não revelados por emuladores. A diferença reside no contexto, dado que FPGAs de núcleo apresentam características distintas dos *switches* comerciais de borda baseados em ASIC. O princípio documentado pelos autores, relativo ao comportamento assíncrono nas operações de escrita e exclusão de regras, fundamenta o fenômeno das regras órfãs observado nesta pesquisa.

Muqaddas *et al.* [21] contribuem com uma perspectiva complementar, ao investigarem o

overhead de comunicação entre controladores em *clusters* ONOS para a manutenção da consistência via Atomix/Raft. Os dados indicam que, mesmo na ausência de ataques, a sincronização de estado no ONOS introduz latências mensuráveis. Sob ataque volumétrico, com o canal de controle congestionado, essas latências se amplificam, podendo levar o controlador a considerar removida uma regra que o *switch* ainda não processou. Mohammed [35] reforça esse diagnóstico ao identificar vulnerabilidades na gestão de estado distribuído do ONOS, o que motivou a implementação do *State Checker* como mecanismo externo de auditoria nesta dissertação.

A escassez da memória TCAM, central ao fenômeno investigado nesta dissertação, é também o alvo de uma classe de ataques recentes. Cao *et al.* [36], com o ataque LOFT, demonstraram que a tabela de fluxos de um comutador OpenFlow pode ser exaurida a baixa taxa e de forma furtiva, o que evidencia a TCAM como recurso fisicamente limitado e operacionalmente crítico. O problema tratado neste trabalho tem origem distinta, pois não decorre de um esgotamento provocado pelo atacante, e sim da persistência não intencional de regras após a remoção lógica, mas compartilha a mesma raiz física, a saber, a escassez e a latência de escrita da TCAM em *hardware* comercial. Na direção da defesa, Chen *et al.* [37], com o SDNShield, propuseram uma estrutura baseada em virtualização de funções de rede (NFV) para conter a saturação do plano de controle por tempestades de `PACKET_IN`, a mesma pressão modelada no cenário de ameaça desta pesquisa, ainda que por um caminho de solução diverso do verificador em malha fechada aqui adotado. Ding *et al.* [38] avançaram na identificação de vítimas de inundação volumétrica diretamente no plano de dados de comutadores comerciais programáveis, reforçando a tendência de levar a inteligência de defesa para a borda, embora também sob a premissa de *hardware* programável. Uma visão sistemática dessas ameaças e contramedidas, que abrange o esgotamento de TCAM, o *overflow* de tabelas e a saturação do canal de controle, é oferecida pelo levantamento recente de Tang *et al.* [39], que situa o problema de consistência e de persistência de regras no panorama atual da segurança em redes programáveis.

2.5.5 Síntese das lacunas identificadas

A Tabela 2.3 reúne os trabalhos discutidos nas subseções anteriores em formato comparativo, utilizando cinco critérios metodológicos: técnica ou abordagem adotada; ambiente de validação; uso de *hardware* físico real; investigação de regras órfãs; e principal limitação identificada.

A comparação evidencia um padrão consistente. Entre os trabalhos validados em equipamento físico, Wijeratne *et al.* [9] restringiram-se a FPGAs de núcleo, e a linha de defesa no plano de dados programável (Poseidon, Jaqen, Euclid e INDDoS) opera sobre comutadores P4 de função programável. Nenhuma dessas categorias corresponde ao comutador OpenFlow comercial de função fixa avaliado nesta dissertação, cuja restrição a um *pipeline* de tabela única define a condição operacional aqui investigada. O discriminador decisivo está na coluna de regras órfãs: o fenômeno não é tratado por nenhum dos estudos listados. Tal ausência não implica a inexistência do problema nos ambientes analisados, mas indica que nem os emuladores nem os comutadores P4 de função programável expõem a defasagem de escrita e remoção da TCAM observada no

hardware comercial de função fixa.

Essa lacuna possui implicações práticas diretas: um mecanismo de defesa SDN que opera de forma satisfatória em simulação pode falhar silenciosamente em ambiente de produção, bloqueando tráfego legítimo de modo indefinido em razão de regras que o controlador considera removidas, mas que o *switch* físico mantém ativas no plano de dados. Esse é o problema posicionado no centro da investigação desta dissertação e que, até onde a revisão da literatura permitiu identificar, não havia sido documentado e resolvido empiricamente em *hardware* comercial.

Tabela 2.3: Comparação sistemática dos principais trabalhos relacionados em segurança SDN para ambientes IoT.

Trabalho	**Técnica / Abordagem**	**Ambiente**	**HW Real**	**Reg. Órfãs**	**Limitação Principal**
Braga *et al.* (2010) [24]	Métricas OpenFlow / NBA (NOX)	Emulado	Não	Não	Sem validação em *hardware* real; limiares fixos
Giotis *et al.* (2014) [25]	OpenFlow + sFlow (anomalia)	Emulado	Não	Não	Sem análise de TCAM; sem *switch* físico
Scott-Hayward *et al.* (2016) [3]	*Survey* segurança SDN	Teórico	Não	Não	Sem implementação; análise conceitual
Tootoonchian *et al.* (2012) [10]	Desempenho de controladores SDN	Emulado	Não	Não	Sem foco em segurança IoT
Meidan *et al.* (2018) [23]	*Deep autoencoder* (N-BaIoT)	*Dataset*	Não	Não	SDN não integrado ao modelo
Wijeratne *et al.* (2019) [9]	SDN em FPGA (redes de núcleo)	FPGA (parcial)	Parcial	Não	Sem segurança IoT; sem ASIC comercial
Shinan *et al.* (2021) [8]	Revisão ML em SDN	Revisão sist.	Não	Não	Sem impl.; latência de inferência ignorada
Aslam *et al.* (2024) [6]	ONOS DDoS Defender	Emulado	Não	Não	Sem *hardware* físico; sem regras órfãs
Romani *et al.* (2025) [32]	Aprendizado Federado DDoS IoT	Simulado	Não	Não	Sem HIL; *overhead* FL não avaliado
Souza e Arima (2024) [11]	Detecção malware IoT/SDN (ONOS)	Emulado	Não	Não	Sem análise de TCAM física
Poseidon[26], Jaqen[27], Euclid[29], INDDoS[38] (2020–2021)	Defesa/detecção DDoS *in-network* no plano de dados programável (P4)	*Switch* P4 / Tofino	Sim (P4)	Não	Pressupõem *switch* P4/Tofino programável; não tratam regras órfãs
Esta pesquisa (2026)	**DDSO + State Checker (HIL)**	**HW real**	**Sim**	**Sim**	Única família de ASIC; limiares fixos

3. PROPOSTA DE ARQUITETURA E AMBIENTE EXPERIMENTAL

Este capítulo detalha a concepção, a estruturação e a implementação da arquitetura de defesa orientada por SDN proposta nesta dissertação. Diferentemente de abordagens baseadas estritamente em emuladores de rede [15, 8], o desenho metodológico aqui apresentado adota o modelo *Hardware-in-the-Loop* (HIL), integrando um *switch* comercial físico ao plano de controle e às fontes de tráfego virtualizadas. Essa escolha visa garantir que as soluções de mitigação sejam validadas sob as restrições reais de I/O e de memória de um equipamento de produção, e não apenas sob as abstrações de um ambiente puramente emulado.

O capítulo está organizado da seguinte forma. A Seção 3.1 descreve a topologia da rede experimental, as especificações do *switch bare-metal* comercial (Dell S3048-ON) e a sua comunicação via protocolo OpenFlow com o controlador ONOS [20], além do provisionamento dos recursos físicos no *hypervisor* Proxmox, da segmentação por VLANs, do papel do *firewall* de perímetro na contenção do tráfego experimental, da extensão multi-switch do *testbed* e da reescrita de endereço MAC no salto de camada 3. A Seção 3.2 aborda a modelagem do cenário de ameaça, especificando as ferramentas e os parâmetros utilizados para a geração de tráfego sintético da família Mirai. A Seção 3.3 apresenta a instrumentação de observabilidade e coleta de métricas. Por fim, a Seção 3.4 detalha o núcleo lógico da proposta: o módulo de segurança DDSO (*DDoS SDN Orchestrator*), os algoritmos heurísticos de detecção codificados em Python, a comunicação via *Northbound API* RESTful com o ONOS e o mecanismo de verificação de consistência em malha fechada (*State Checker*), que constitui a principal contribuição de engenharia deste trabalho.

3.1 TOPOLOGIA DA REDE EXPERIMENTAL E PROVISIONAMENTO DE HARDWARE

A operacionalização do modelo HIL exigiu a separação rigorosa entre os planos de controle e de dados, com a rede de gerência conduzida fora de banda (*out-of-band management*). A natureza *Hardware-in-the-Loop* desta pesquisa reside na presença do *switch* comercial Dell S3048-ON, físico, no caminho dos dados, com sua memória TCAM, seu ASIC de comutação e seu comportamento assíncrono de *hardware*, condições ausentes em validações puramente emuladas. As demais entidades do experimento (os nós atacantes, o alvo e o controlador) foram implementadas como máquinas virtuais (VMs) sobre o *hypervisor* Proxmox. A Figura 3.1 apresenta a arquitetura sistêmica completa, organizada de modo a destacar a interação entre a inteligência computacional e a infraestrutura física. A figura evidencia os três planos de operação, o de Aplicação (DDSO), o de Controle (ONOS) e o de Dados Físico (Dell S3048-ON), e os protocolos de comunicação entre as camadas.

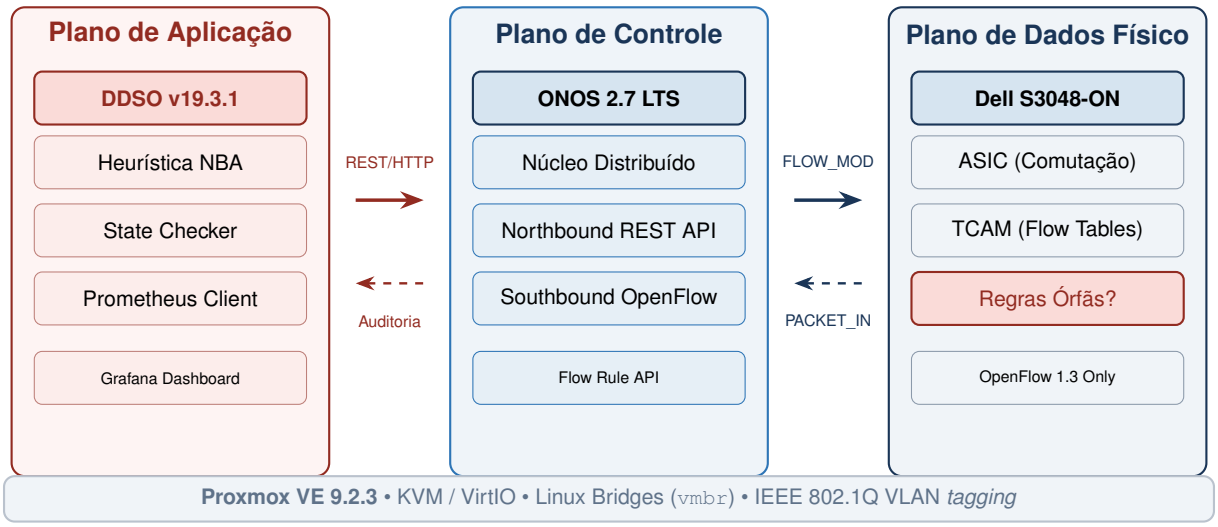

Figura 3.1: Arquitetura de defesa sistêmica em modelo *Hardware-in-the-Loop* (HIL).

O elemento central do plano de dados nesta topologia é o *switch* Dell Networking S3048-ON. Trata-se de um equipamento de borda (*Top-of-Rack*) do tipo *Open Networking*, que adota o padrão ONIE para a instalação do sistema operacional de rede e executa o Dell SmartFabric OS10, sobre um ASIC de comutação comercial [40, 41, 42]. A escolha deste equipamento é central para a metodologia HIL, pois ele impõe ao experimento as restrições reais da indústria: a memória TCAM disponível para a alocação de fluxos OpenFlow 1.3 é estritamente limitada, e a propagação de regras entre o controlador e o plano de dados está sujeita ao comportamento assíncrono de escrita e remoção em *hardware* [9].

A adoção do ONOS como plano de controle decorreu de uma avaliação prévia de compatibilidade entre o agente OpenFlow do *switch* e os controladores SDN candidatos. Embora a especificação OpenFlow 1.3 preveja o encadeamento de múltiplas tabelas de fluxo (*pipeline processing*), o agente do OS10 expõe ao controlador um modelo de *pipeline* de tabela única (Tabela 0), no qual a totalidade das entradas de fluxo é alocada, conforme verificado na interface de gerenciamento do ONOS. Essa característica é determinante para a seleção do controlador: o OS10 implementa um subconjunto restrito da especificação OpenFlow 1.3, rejeitando, com mensagens `OFPT_ERROR`, ações de modificação de campos de cabeçalho e modos de encaminhamento como `OUTPUT:NORMAL` e `OUTPUT:TABLE` [43].

Em razão dessa restrição, a tentativa de integração com o controlador Faucet, que pressupõe suporte pleno a múltiplas tabelas OpenFlow no plano de dados, não resultou na instalação de fluxos no equipamento, com nenhuma regra efetivada no *switch* [44, 45]. Avaliou-se também a substituição do sistema operacional de rede pelo Cumulus Linux. Dois fatores descartaram essa alternativa. O primeiro foi a descontinuação (*End of Life*) do suporte à linha de equipamentos de 1 Gbps à qual pertence o S3048-ON, no contexto da migração do produto para um modelo licenciado sob a NVIDIA [46]. O segundo foi a incompatibilidade de paradigma: a adoção do Cumulus implicaria abandonar o controle centralizado via OpenFlow, fundamento do módulo DDSO, em favor de um modelo de rede desagregada baseado em protocolos distribuídos. Considerou-se, ainda, o emprego

do conjunto de aplicações Trellis, que executa sobre o ONOS e provê uma malha *leaf-spine* L2/L3 de código aberto. Essa alternativa também se mostrou inviável: o seu mecanismo de roteamento por segmentos (*segment routing* sobre MPLS) pressupõe o *pipeline* OF-DPA de múltiplas tabelas, ausente no agente OpenFlow do OS10 [47]. O ONOS, operando com um *pipeline* de tabela única aderente ao subconjunto OpenFlow exposto pelo OS10, mostrou-se a plataforma compatível com o equipamento, comportamento igualmente reportado para controladores baseados em Ryu [45].

A rede foi dividida em zonas operacionais para garantir a fidelidade e a contenção dos testes:

- **Zona de Alvo:** composta pela máquina `vm-target-h-01` (164.72.55.1/24) na VLAN 55, deliberadamente mantida sem *hardening* e configurada para expor um conjunto de serviços vulneráveis em portas bem conhecidas (entre eles HTTP, Telnet e SSH), de modo a reproduzir a superfície de exposição característica dos dispositivos visados pela *botnet* Mirai e permitir a avaliação do esgotamento de recursos.
- **Zona de Comando e Controle (C2):** onde reside o controlador da *botnet* Mirai (`vm-mirai-cnc-h-01`, 172.16.23.1) na VLAN 23, responsável por despachar as ordens de inundação.
- **Zona de Ataque (Bots):** composta por três máquinas virtuais (`vm-bot-h-01` a `h-03`) na VLAN 22 (164.72.22.0/24), atuando como nós da *botnet* distribuída, à qual a extensão multi-switch acrescentou um quarto nó remoto (Seção 3.1.4). A geração de tráfego foi virtualizada; nós físicos do tipo Raspberry Pi, já presentes na infraestrutura, ficam registrados como extensão futura no Capítulo de Conclusão.

A Tabela 3.1 detalha os ativos do ambiente experimental, incluindo os incorporados pela extensão multi-switch.

Tabela 3.1: Detalhamento dos ativos do ambiente experimental (HIL), incluindo a extensão multi-switch.

Hostname	**IP/CIDR**	**VLAN**	**Função / Serviço**	**SO / Hardware**
sw-sdn-p-01	172.16.9.11/24	9	*Switch* SDN OpenFlow (SW-A)	Dell S3048-ON (físico)
sw-sdn-p-02†	172.16.9.12/24	9	*Switch* SDN OpenFlow (SW-B)	Dell S3048-ON (físico)
vm-onos-p-01	172.16.9.15/24	9	Controlador SDN (ONOS)	Linux (VM)
vm-target-h-01	164.72.55.1/24	55	Máquina alvo (vulnerável)	Linux (VM)
vm-mirai-cnc-h-01	172.16.23.1/24	23	Servidor C2 (*Mirai Controller*)	Linux (VM)
vm-bot-h-01	164.72.22.1/24	22	Nó *botnet* (gerador de tráfego)	Linux (VM)
vm-bot-h-02	164.72.22.2/24	22	Nó *botnet* (gerador de tráfego)	Linux (VM)
vm-bot-h-03	164.72.22.3/24	22	Nó *botnet* (gerador de tráfego)	Linux (VM)
vm-bot-h-04†	164.72.22.4/24	22	Nó *botnet* remoto (sobre SW-B)	Linux (VM)
vm-dns-h-01	164.72.12.25/24	12	Servidor DNS	Linux (VM)
vm-dash-h-01	164.72.11.1/24	11	*Dashboard* (Grafana/Prometheus)	Linux (VM)
vm-akvorado-p-01	164.72.30.1/30	30	Observabilidade (sFlow/NetFlow)	Linux (VM)
vm-opnssense-g-01	(*trunk*)	—	*Firewall* / roteamento *inter*-VLAN	OPNsense

† Ativos incorporados na extensão multi-switch (Seção 3.1.4). Na campanha do vetor GRE (Capítulo 4), o SW-B participou da sincronização da regra de descarte nos dois comutadores, e o `vm-bot-h-04` serviu à demonstração da janela de indisponibilidade reativa (Seção 3.1.4.1); o ataque GRE, como os demais, partiu dos *bots* `h-01` a `h-03` sobre o SW-A.

3.1.1 Segmentação por *Firewall* e Contenção do Tráfego Experimental

A separação entre as zonas operacionais e a contenção do tráfego malicioso foram asseguradas por um *firewall* OPNsense (`vm-opnssense-g-01`), responsável pelo roteamento *inter*-VLAN do ambiente. O equipamento concentra as interfaces das VLANs do experimento (entre elas a VLAN 22 dos *bots*, a VLAN 55 do alvo e a VLAN 23 do C2) e aplica políticas de *stateful firewall* que restringem a comunicação entre os segmentos ao estritamente necessário para a reprodução do ataque, impedindo o vazamento do tráfego sintético para fora do laboratório. Adicionalmente, o *firewall* foi configurado para exportar amostras de fluxo (NetFlow v5) para a plataforma de observabilidade Akvorado, o que permitiu a inspeção volumétrica do tráfego independentemente da instrumentação do orquestrador.

Essa política de *stateful firewall* permaneceu ativa em todas as condições experimentais, inclusive na condição de referência (*off*), o que é essencial para a correta interpretação dos resultados do Capítulo 4. A condição de referência corresponde, portanto, a um ambiente protegido pelo *firewall* de perímetro com o orquestrador DDSO desativado, e não a um ambiente sem qualquer mitigação. Diante de origem estável, a filtragem de estado do *firewall* exerce contenção parcial, o que reduz a exposição de referência dos vetores SYN e UDP em relação à duração nominal do disparo; as reduções atribuídas ao DDSO são, assim, incrementais em relação a essa contenção de perímetro. Sob falsificação dinâmica de origem, como no vetor ACK, a filtragem de estado por endereço é evadida, e a exposição de referência aproxima-se da duração integral do disparo.

A Figura 3.2 apresenta a topologia lógica e física do ambiente, já com a extensão multi-switch. O comutador `sw-sdn-p-01` (SW-A) concentra o controlador, o alvo, a *botnet* local e a observabilidade; o comutador `sw-sdn-p-02` (SW-B), interligado a SW-A por um enlace de 1 GbE na porta 1/1/33 (porta lógica 129 no ONOS), hospeda o *bot* remoto `vm-bot-h-04`. O *firewall* OPNsense, conectado por uma porta *trunk* 802.1Q, atua como roteador L3 entre as VLANs e exporta *NetFlow* para o Akvorado. O SW-B e o `vm-bot-h-04` foram incorporados após as campanhas dos vetores SYN, ACK e UDP, estendendo o ambiente a um domínio de encaminhamento de dois comutadores. Na campanha do vetor GRE (Capítulo 4), conduzida a partir dos *bots* locais sobre o SW-A, a regra de descarte foi sincronizada nos dois comutadores; o `vm-bot-h-04` foi empregado na demonstração da janela de indisponibilidade reativa entre comutadores distintos (Seção 3.1.4.1). A segmentação *out-of-band* isola os planos de controle (VLAN 9), de dados/alvo (VLAN 55), de observabilidade (VLAN 30), de DNS (VLAN 12) e de ataque (VLANs 22 e 23).

3.1.2 Configuração do *Hypervisor* e Virtualização de Rede

Para sustentar a arquitetura lógica e garantir o isolamento sem penalizar o roteamento de pacotes, adotou-se o *Proxmox Virtual Environment* (PVE) [48]. A Tabela 3.2 detalha as especificações deste *testbed*.

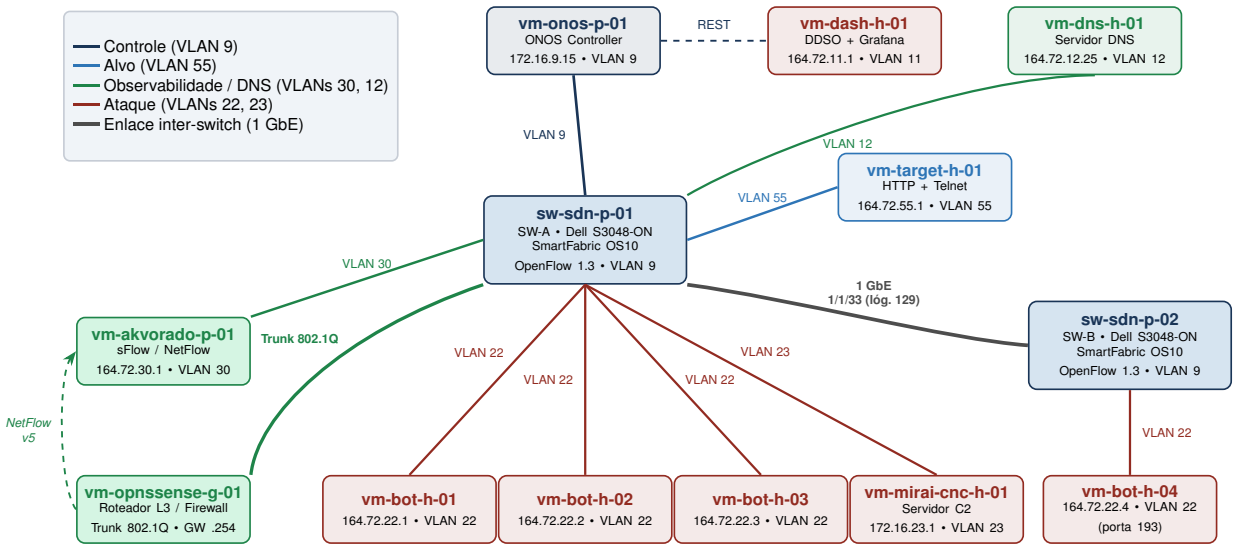

Figura 3.2: Topologia lógica e física do ambiente *Hardware-in-the-Loop* com a extensão multi-switch.

Tabela 3.2: Especificações técnicas do servidor hospedeiro (Proxmox VE).

Componente	**Especificação Técnica**
Hypervisor	Proxmox Virtual Environment (PVE) versão 9.2.3
Kernel do SO	Linux 6.17.9-1-pve
Processador	2x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz (32 *Threads*)
Memória RAM	128 GB (125.79 GiB utilizáveis)
Armazenamento	ZFS / LVM-Thin
Virtualização	KVM com *QEMU Guest Agent* e *Drivers VirtIO*

A Tabela 3.3 detalha a alocação de recursos computacionais atribuída a cada máquina virtual do ambiente, complementando as especificações do servidor hospedeiro.

Tabela 3.3: Alocação de recursos computacionais por máquina virtual no ambiente *Hardware-in-the-Loop*, conforme configurados no *hypervisor* Proxmox VE. Os quatro nós *bot* possuem configuração idêntica.

Função (VM)	**vCPU**	**RAM**	**Disco**
Controlador ONOS (`vm-onos-p-01`)	8	16 GiB	32 GB
Alvo (`vm-target-h-01`)	4	20 GiB	200 GB
Servidor C2 Mirai (`vm-mirai-cnc-h-01`)	4	4 GiB	50 GB
Nós *bot* (`vm-bot-h-01` a `-04`)	4	4 GiB	50 GB
Dashboard Grafana/Prometheus (`vm-dash-h-01`)	4	8 GiB	100 GB
Servidor DNS (`vm-dns-h-01`)	4	2 GiB	50 GB
Observabilidade Akvorado (`vm-akvorado-p-01`)	16	16 GiB	200 GB
Firewall/Roteador L3 OPNsense (`vm-opnssense-g-01`)	8	16 GiB	32 GB

A integração entre as máquinas virtuais hospedadas no Proxmox e o *switch* físico SDN Dell exigiu a configuração de *Linux Bridges* (`vmbr`). As interfaces físicas do servidor foram associadas às *bridges*, garantindo que o tráfego gerado pelas instâncias virtuais fosse injetado diretamente nas portas físicas correspondentes, com as marcações de VLAN (*tagging* IEEE 802.1Q) intactas. A utilização de *drivers* paravirtualizados (*VirtIO*) minimizou o *overhead* do hipervisor, reduzindo a contribuição da camada de virtualização sobre as latências medidas [48]. A capacidade do servidor

hospedeiro foi dimensionada para a folga operacional dos ensaios, e não constitui requisito para a reprodução dos resultados. O elemento determinante do experimento é o *switch* físico Dell S3048-ON no caminho dos dados, cujo comportamento assíncrono de *hardware* é o objeto de estudo; as máquinas virtuais apenas geram e recebem tráfego. Como as métricas de tempo de exposição e de latência de mitigação são governadas pelo comutador físico e pelo período de varredura do orquestrador, e não pela capacidade do hospedeiro, o ambiente é reproduzível em plataformas de menor porte, desde que preservados o comutador físico e a segmentação de rede descritos nesta seção. Ressalva-se, contudo, que o gerador de tráfego, o alvo e o controlador compartilham o mesmo servidor hospedeiro, de modo que as métricas reportadas refletem o comportamento do conjunto formado pelo comutador físico, pelo hipervisor e pelas máquinas virtuais. As métricas de contenção de recursos no hospedeiro, entre elas o uso de CPU, o processamento de interrupções, as filas das interfaces e os descartes nas *bridges* virtuais, não foram instrumentadas de forma sistemática, limitação retomada na Seção 5.4.

3.1.3 Configuração do controlador ONOS

O plano de controle foi implementado sobre o ONOS na versão 2.7 LTS, indicada na Figura 3.1, operando em configuração enxuta: mantiveram-se ativas apenas as aplicações estritamente necessárias ao experimento, a saber, `drivers`, `openflow-base`, `openflow`, `hostprovider`, `lldpprovider`, `netcfghostprovider`, `gui2`, `fwd` (*Reactive Forwarding*) e `proxyarp`. A restrição do conjunto de aplicações reduz fontes de interferência sobre as medições e mantém o controlador em estado reproduzível.

Um parâmetro de configuração é tratado como constante metodológica: a frequência de sondagem de estatísticas de fluxo do provedor OpenFlow (`flowPollFrequency`) foi fixada em 1 segundo e mantida inalterada em todas as campanhas. Esse período governa o instante em que os contadores atualizados tornam-se visíveis ao orquestrador e, em conjunto com o intervalo de telemetria do DDSO ($\texttt{INTERVAL} = 1$ s, Seção 3.4), sustenta o regime de latência de mitigação reportado no Capítulo 4. Qualquer alteração desse valor comprometeria a comparabilidade das medições de latência.

A opção pelo ONOS executando sobre *hardware* físico é deliberada: as alternativas baseadas no controlador Ryu e na comutação por *software* Open vSwitch, disponíveis no laboratório, permaneceram desativadas durante todas as campanhas, de modo a isolar o comportamento do controlador adotado sobre comutação em silício real. Essa decisão é coerente com a análise de compatibilidade de plataforma apresentada nesta seção, que descartou as pilhas Faucet, Cumulus e Trellis em razão do *pipeline* de tabela única exposto pelo agente OpenFlow do OS10.

3.1.4 Extensão multi-switch do testbed

A Figura 3.2 apresenta o ambiente já com a extensão multi-switch. As campanhas dos vetores SYN, ACK e UDP, reportadas no Capítulo 4, foram conduzidas sobre um único comutador físico

(sw-sdn-p-01, SW-A); a inclusão de um segundo comutador visa avaliar o comportamento do plano de controle diante de um caminho de encaminhamento distribuído entre dois dispositivos, condição representativa de uma borda real com múltiplos saltos, e sustenta a campanha do vetor GRE.

O comutador original, doravante referido como SW-A (sw-sdn-p-01, DPID of:00003c2c3024ef80, gerência 172.16.9.11), permanece responsável por servir o alvo (vm-target-h-01) e a *botnet* local (vm-bot-h-01 a -03). O segundo comutador, SW-B (sw-sdn-p-02, DPID of:00003c2c30250880, gerência 172.16.9.12), foi incorporado para hospedar um *bot* remoto. Os dois equipamentos são interconectados por um enlace de 1 GbE entre as portas 1/1/33 de cada um, constituindo um único domínio de encaminhamento gerenciado pelo ONOS.

O *bot* remoto vm-bot-h-04 (164.72.22.4, VLAN 22) é o único hospedeiro conectado ao SW-B, na porta 193, e foi empregado na demonstração da janela de indisponibilidade reativa entre comutadores (Seção 3.1.4.1). Cumpre registrar que tanto o SW-B quanto esse *bot* foram incorporados após as coletas dos vetores SYN, ACK e UDP, as quais utilizaram exclusivamente os *bots* h-01 a h-03 sobre o SW-A. A extensão, portanto, não retroage sobre esses resultados: ela amplia o ambiente para a campanha do vetor GRE, de origem real distribuída a partir dos *bots* locais sobre o SW-A, na qual a sincronização da regra de bloqueio foi registrada nos dois comutadores (Capítulo 4).

3.1.4.1 Janela de indisponibilidade reativa

A introdução de um segundo salto físico tornou observável um efeito diretamente relacionado à tese de persistência de regras desta dissertação. Verificou-se que a primeira tentativa de comunicação "a frio" entre hospedeiros situados em comutadores distintos (por exemplo, o estabelecimento inicial de uma sessão SSH) sofre perda dos pacotes iniciais. O comportamento decorre da aplicação *Reactive Forwarding* (fwd) do ONOS, que pavimenta o caminho de encaminhamento de modo reativo, instalando *uma entrada de fluxo por salto* somente após o primeiro pacote ser elevado ao controlador via OFPT_PACKET_IN. Durante o intervalo de instalação dessas regras, os pacotes iniciais não encontram correspondência no plano de dados e são descartados.

A documentação oficial do ONOS confirma o mecanismo de provisionamento salto a salto [49]. O achado reforça o argumento central do trabalho: a dependência de regras provisionadas reativamente cria uma janela transitória de indisponibilidade, que tende a se agravar à medida que o número de saltos aumenta, motivando estratégias de persistência e sincronização de regras na borda.

3.1.5 Reescrita de endereço MAC no salto de camada 3

O tráfego de ataque originado nos *bots* (VLAN 22) e destinado ao alvo (VLAN 55) atravessa uma fronteira de camada 3: o roteamento *inter*-VLAN é realizado pelo *firewall* OPNsense (`vm-opnssense-g-01`), descrito na Seção 3.1.1. Essa fronteira tem consequência direta sobre a seleção da âncora de bloqueio empregada pelo módulo DDSO e foi verificada empiricamente por captura simultânea com `tcpdump` em dois pontos do caminho.

No ponto de ingresso, na perna do OPNsense voltada à VLAN 22, o endereço MAC de origem observado corresponde ao do *bot* real (`vm-bot-h-04`, `BC:24:11:F1:27:04`). No ponto de egresso, no segmento da VLAN 55 junto ao alvo, o endereço MAC de origem observado já é o do próprio *firewall* (`BC:24:11:8B:48:F4`): o endereço físico do *bot* desaparece, substituído pelo do roteador no salto L3. O endereço IP de origem, em contrapartida, é preservado ao longo de todo o caminho, pois não há tradução de endereços (NAT) configurada.

A implicação para o projeto do detector é decisiva. A âncora de bloqueio baseada no endereço físico de origem (`ETH_SRC`) só é válida no segmento de ingresso, anterior ao salto L3, onde o MAC do *bot* ainda está presente. No segmento de egresso, a mesma âncora seria não apenas inócua como nociva: uma regra ancorada em `ETH_SRC` nesse ponto corresponderia ao MAC do *firewall* e bloquearia indiscriminadamente todo o tráfego *inter*-VLAN legítimo que transita pelo *gateway*.

Esse resultado fundamenta a exigência de consciência de topologia no DDSO e justifica a estratégia de ancoragem adaptativa consolidada na versão final do orquestrador (v19.3.1, Apêndice B): o bloqueio parte do endereço IP de origem (`IPV4_SRC`), preservado fim a fim e, portanto, robusto à reescrita de MAC, escalando para `ETH_SRC` apenas quando o cenário de *spoofing* o exige e a regra é aplicada no ponto de ingresso correto. A âncora física é, em suma, posicionalmente dependente: eficaz a montante do salto L3 e contraindicada a jusante.

3.2 MODELAGEM DO CENÁRIO DE AMEAÇA E EXAUSTÃO DE ESTADO

A validação da arquitetura exigiu a reprodução de um ataque volumétrico orquestrado. A modelagem foi estruturada por meio do C2 (`vm-mirai-cnc-h-01`), que despacha diretivas de ataque para os *bots* virtualizados na VLAN 22. O modelo de ameaça toma como referência a inundação SYN distribuída [4], configurada para explorar duas frentes de pressão simultâneas:

- **Exaustão do Alvo (*Endpoint Starvation*):** inundação de requisições incompletas na porta 80 do alvo (`vm-target-h-01`) para esgotar a alocação de *sockets* TCP e o *backlog* de conexões pendentes.
- **Tempestade de *Packet-In* (*Control Plane Exhaustion*):** uso de *IP Spoofing* dinâmico [50]. Ao gerar pacotes com endereços de origem forjados e aleatórios, o ataque provoca sucessivos eventos de *table-miss* no Dell S3048-ON. Cada novo pacote forjado obriga o *switch* a encaminhar o respectivo cabeçalho ao ONOS por meio de mensagens `OFPT_PACKET_IN`.

Essas duas frentes de pressão são instanciadas, com parâmetros controlados, nos quatro vetores avaliados no Capítulo 4: as inundações SYN e UDP de origem fixa, a inundação ACK com falsificação de origem e a inundação GRE de origem real distribuída.

A Figura 3.3 apresenta o fluxograma dessa orquestração, ilustrado para o vetor de referência, a inundação SYN, com a resposta do orquestrador DDSO em modo IPS.

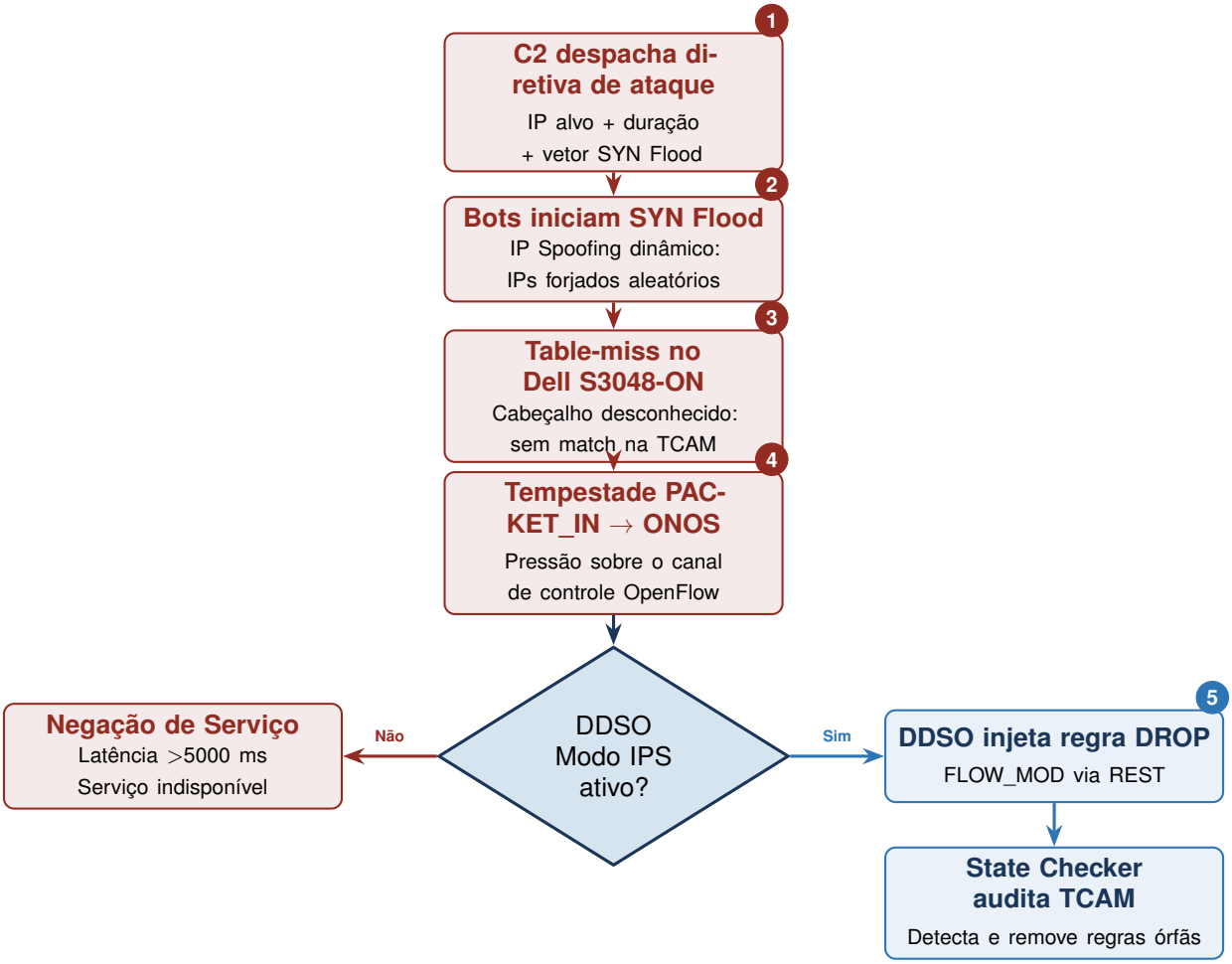

Figura 3.3: Fluxograma de orquestração do ataque volumétrico *Mirai-like* e resposta do orquestrador DDSO.

Quando a taxa de chegada de pacotes desconhecidos supera a capacidade do canal de controle OpenFlow em escoá-los, a propagação de regras entre o controlador e o *switch* passa a competir com o próprio tráfego de *table-miss*. É essa pressão sobre o plano de controle que o orquestrador de segurança precisa conter, injetando regras de bloqueio antes que a tempestade comprometa a sincronização entre os planos.

3.3 OBSERVABILIDADE E COLETA DE MÉTRICAS VIA PROMETHEUS

Em ambientes HIL, a extração de métricas exige o provisionamento de ferramentas de telemetria dedicadas. A arquitetura de observabilidade abrange da inspeção de volumetria aos painéis

analíticos:

- **Plataforma Akvorado:** instanciada para a inspeção de volumetria a partir de exportações de *sFlow/NetFlow* provenientes do *firewall* de perímetro, permitindo a visualização das assinaturas de tráfego originadas dos nós atacantes.
- **Integração Prometheus e Grafana:** para capturar o comportamento dinâmico da mitigação SDN, o orquestrador foi instrumentado com a biblioteca `prometheus_client`. Métricas como `sdn_pkts_rate` (taxa de pacotes por IP) e `sdn_mitigation_latency_ms` (tempo, em milissegundos, para a aplicação do bloqueio) foram exportadas em formato *Gauge* para um servidor local na porta 8000 e plotadas no Grafana (`vm-dash-h-01`), compondo o ambiente de análise em tempo real.

3.4 MÓDULO DDSO (DDOS SDN ORCHESTRATOR) E LÓGICA DE MITIGAÇÃO

A materialização da estratégia de defesa descrita nesta dissertação deu-se com o desenvolvimento do módulo *DDoS SDN Orchestrator* (DDSO v19.3.1). Em contraste com aplicações que operam em Java nativamente acopladas ao núcleo do ONOS [20], o DDSO foi projetado como uma aplicação externa codificada em Python. Essa escolha confere maior resiliência e isolamento de falhas: caso o orquestrador sofra interrupções ou sobrecarga no cálculo das heurísticas, o núcleo do ONOS permanece estável operando o roteamento da rede. A versão empregada em todos os ensaios reportados nesta dissertação é a v19.3.1. O orquestrador partiu do protótipo experimental v16.1 e evoluiu de forma iterativa; o Apêndice A apresenta o pseudocódigo de alto nível do ciclo de operação em malha fechada, e o Apêndice B documenta o código-fonte integral dessa versão final, que incorpora a ancoragem adaptativa por `ETH_SRC` descrita no Capítulo 4.

O DDSO interage com o plano de controle por meio de chamadas RESTful (*Northbound API*) [20]. O ciclo de operação em malha fechada (*Closed-Loop*) processa a segurança em três estágios fundamentais:

1. **Telemetria e Heurística Estatística:** utilizando a requisição `GET /onos/v1/flows`, o *script* em Python consolida os contadores OpenFlow (*packets* e *bytes*) agrupados por endereço MAC e IP de origem. Calculando o *delta* em intervalos definidos (neste trabalho, `INTERVAL` $= 1$ segundo, valor que governa a latência de mitigação reportada no Capítulo 4), o sistema determina as taxas instantâneas (*pkts_rate*).
2. **Injeção Reativa (Modo IPS):** quando um endereço de origem excede os limiares definidos (por exemplo, a constante `FLOOD_MIN_PKTS_RATE = 10000`), o DDSO formula um *payload* JSON contendo os critérios `IPV4_SRC` e a instrução vazia (equivalente à ação *Drop* no OpenFlow). A mitigação é imediatamente injetada no *switch* físico via `POST /onos/v1/flows/{deviceId}`.

3. **Verificação de Consistência em Malha Fechada:** devido à latência assíncrona da memória TCAM, o DDSO mantém um rastreamento estrito de todos os bloqueios ativos (`ACTIVE_BLOCKS`). Se o tráfego do IP agressor zerar durante ciclos consecutivos (`MITIGATION_ZERO_RATE_CYCLES = 35`), o orquestrador solicita a purga da regra.

O fragmento de código a seguir (Listagem 3.1) reproduz um trecho literal do núcleo do algoritmo de mitigação, evidenciando o momento em que a heurística detecta a anomalia e dispara a injeção da regra, bem como a rotina de auditoria de regras órfãs. O código-fonte integral da versão consta no Apêndice B.

Listagem 3.1: Trecho do núcleo lógico de mitigação e verificação de estado do orquestrador DDSO (v19.3.1); código-fonte integral no Apêndice B.

```
def manage_mitigation(stats):
    global ACTIVE_BLOCKS
    action_logs = []

    # 1. Avaliacao Heuristica e Injecao Reativa
    for origin_key, s in stats.items():
        if classify_flood(s) or classify_brute(s):
            if origin_key not in ACTIVE_BLOCKS:
                flow_ids = drop_attack(s['ip_src'], s['mac_src'])
                if flow_ids:
                    ACTIVE_BLOCKS[origin_key] = {
                        "flow_ids": flow_ids,
                        "zero_rate_count": 0
                    }

    # 2. Verificacao de Estado (Malha Fechada)
    sources_to_remove = []
    for origin_key, state in ACTIVE_BLOCKS.items():
        if stats.get(origin_key, {}).get("pkts_rate", 0) <= 1:
            state["zero_rate_count"] += 1
            if state["zero_rate_count"] >= MITIGATION_ZERO_RATE_CYCLES:
                undrop_attack(state["flow_ids"])
                sources_to_remove.append(origin_key)
        else:
            state["zero_rate_count"] = 0

    for origin_key in sources_to_remove:
        del ACTIVE_BLOCKS[origin_key]

    return action_logs
```

Na prática, é comum que a API do ONOS responda com o *status* HTTP 204 (Sucesso) à solicitação `DELETE`, sem que a exclusão da entrada seja imediatamente efetivada na TCAM do *switch* Dell. Esse descompasso entre a remoção lógica no controlador e a remoção física no

plano de dados, consistente com o comportamento assíncrono de escrita e remoção em *hardware* documentado por Wijeratne *et al.* [9], gera uma **regra órfã**. O laço de verificação do DDSO contorna esse problema por meio de auditorias contínuas. Caso a regra permaneça ativa no plano de dados e continue incrementando contadores nas estatísticas subsequentes, o orquestrador reemite os comandos de remoção de forma iterativa. A reemissão prossegue até que o *switch* confirme a sincronização de estado, o que restaura a normalidade no *testbed*.

4. ANÁLISE E RESULTADOS

Este capítulo apresenta a avaliação experimental e a análise crítica dos resultados obtidos com a implementação da arquitetura de defesa proposta. O objetivo é demonstrar de forma empírica a eficácia do controlador ONOS [20] na mitigação de ataques volumétricos *Mirai-like* em um ambiente *Hardware-in-the-Loop* (HIL). Ao contrário de métricas obtidas em emuladores, os dados discutidos a seguir refletem o comportamento real e as limitações de um comutador comercial (Dell S3048-ON) [40] sob alto estresse de rede.

A exposição segue a própria sequência da investigação. A primeira seção estabelece a linha de base da rede, caracterizando o tráfego legítimo, o consumo normal de recursos e os tempos de resposta da infraestrutura. A segunda quantifica o impacto da injeção de tráfego malicioso e a eficácia da mitigação sobre a vítima, com base em repetições controladas de um ataque SYN de origem fixa. As duas seções seguintes tratam do cerne da pesquisa empírica, a descoberta dos fenômenos de assincronia de hardware, que produzem regras órfãs no plano de dados do comutador, e a avaliação do mecanismo de verificação de estado que os elimina. As seções finais estendem a avaliação a três vetores adicionais, a inundação ACK com falsificação de endereço de origem, a inundação GRE distribuída de origem real e a inundação UDP de origem fixa, e encerram com a síntese comparativa dos quatro vetores. Os quatro vetores foram avaliados sob período de varredura de um segundo, com trinta repetições por condição, o que torna as latências e os tempos de exposição diretamente comparáveis entre as seções.

4.1 DEFINIÇÃO DA LINHA DE BASE (*BASELINE*) DA INFRAESTRUTURA

A validação de um sistema de detecção de anomalias exige um referencial de normalidade. A linha de base foi extraída em um período de operação sem tráfego malicioso, durante o qual um nó legítimo gerou carga de referência de baixa intensidade contra a máquina alvo por meio do utilitário `iperf3`, com a taxa de transmissão limitada a 3 Mbit/s (`iperf3 -c 164.72.55.1 -b 3M`). Essa carga foi escolhida por permanecer cerca de duas ordens de grandeza abaixo do limiar de detecção de inundação do orquestrador, fixado em 10.000 pacotes por segundo, de modo que o tráfego é contabilizado nas estatísticas, mas classificado como normal e não submetido a qualquer regra de descarte. Em paralelo, a responsividade da camada de aplicação foi aferida por requisições HTTP ao alvo. Durante todo o intervalo, o módulo DDSO operou de forma passiva, limitando-se a coletar métricas pela *Northbound* API do ONOS [20]. A Tabela 4.1 sintetiza os valores médios de operação normal do ambiente HIL.

Tabela 4.1: Métricas de operação normal (*Baseline*) do ambiente *Hardware-in-the-Loop*.

Métrica Analisada	**Valor Médio**	**Desvio Padrão (σ)**
Taxa de Pacotes Legítimos (PPS)	450 pacotes/s	$\pm$ 45
Banda Consumida	3,2 Mbps	$\pm$ 0,4
Uso de Memória TCAM (Fluxos L2/L3)	15 entradas	$\pm$ 0
Latência Média de Aplicação (HTTP RTT)	4,5 ms	$\pm$ 1,2

O baixo volume de tráfego legítimo (em torno de 450 pacotes por segundo), a ocupação reduzida da TCAM (apenas 15 entradas) e a latência de aplicação estável (RTT HTTP de 4,5 ms) caracterizam um ambiente em operação normal, sem gargalos no plano de dados. Esse cenário assegura que as medidas de eficácia e de latência discutidas adiante reflitam o comportamento do mecanismo de defesa, e não condições preexistentes da infraestrutura.

4.2 IMPACTO DO ATAQUE VOLUMÉTRICO E EFICÁCIA DA MITIGAÇÃO

Com o referencial estabelecido, o cenário foi submetido à orquestração de ataque pelo servidor de comando e controle da *botnet*. Três dispositivos infectados, posicionados na VLAN 22 com endereços 164.72.22.1, 164.72.22.2 e 164.72.22.3, iniciaram um ataque coordenado do tipo SYN *flood* contra a máquina alvo `vm-target-h-01`, de endereço 164.72.55.1, situada na VLAN 55. Cada execução foi disparada com o comando `syn 164.72.55.1 60`, que instrui os *bots* a inundar o alvo por sessenta segundos.

Uma característica do vetor empregado merece registro, por orientar a interpretação dos resultados. Os *bots* atacaram a partir de seus endereços reais e fixos, sem falsificação dinâmica de origem. Essa condição é determinante para a estratégia de mitigação por bloqueio de endereço IP de origem, que se mostra plenamente eficaz contra fontes de origem estável, ao passo que vetores com falsificação de origem exigem uma estratégia de ancoragem distinta, cuja avaliação é o objeto da Seção 4.5.

4.2.1 Carga sobre o Plano de Controle

A injeção do tráfego anômalo causou degradação simultânea em duas frentes, a exaustão do alvo e o aumento da carga sobre o plano de controle. À medida que novos fluxos do ataque ingressavam no comutador sem correspondência prévia na memória TCAM (evento de *table-miss*), o equipamento, em conformidade com a especificação OpenFlow [2], encapsulava os cabeçalhos e os encaminhava ao controlador ONOS por mensagens `PACKET_IN`. A taxa de ingresso no alvo ultrapassou rapidamente a marca de cinquenta mil pacotes por segundo, com picos individuais superiores a setenta mil pacotes por segundo, conforme registrado na interface da própria vítima.

Sob essa carga, e sem uma ação de defesa que atuasse sobre a origem do tráfego, a vazão do

ataque consumia a capacidade de atendimento da máquina alvo, degradando a disponibilidade dos serviços legítimos por ela providos. Esse comportamento, embora limite a duração efetiva da inundação pela própria exaustão dos recursos do alvo, representa o colapso da disponibilidade do serviço, e não uma forma de proteção.

4.2.2 Procedimento Experimental e Métricas

A avaliação da mitigação reproduz a estrutura de quatro condições adotada nos demais vetores deste capítulo. Na condição de referência (*off*), o controlador permaneceu em encaminhamento normal, porém com o módulo DDSO inativo, de modo que nenhuma regra de bloqueio fosse instalada durante o ataque. As três condições de mitigação correspondem às estratégias de ancoragem da regra de descarte descritas na Subseção 4.5.2, a saber, o bloqueio por endereço IP de origem (*ip*), o bloqueio por endereço físico de origem (*mac*) e a estratégia adaptativa (*auto*). A repetição das três estratégias em um vetor de origem fixa, em que o bloqueio por IP já é suficiente, tem propósito metodológico. Permite verificar que a estratégia adaptativa, diante de origem confiável, permanece na âncora de camada de rede sem escalonamento, em coerência com o observado nos vetores UDP e GRE.

A telemetria foi capturada na interface `enp6s19` da máquina vítima, que registra o volume efetivamente recebido pelo hospedeiro atacado e constitui a medida mais fiel do impacto sofrido. O sensor registrou os contadores da interface a cada segundo, com marcação temporal absoluta, em uma janela estendida de modo a abranger com folga o disparo e o retorno à normalidade. O controlador operou com período de varredura dos contadores de fluxo de um segundo. Cada condição foi repetida trinta vezes. Duas métricas orientam a análise. O tempo de exposição do alvo corresponde ao número de segundos em que a interface da vítima registrou taxa de pacotes acima do limiar de inundação, e traduz a eficácia da defesa. A latência de mitigação, aplicável às condições de mitigação, corresponde ao intervalo entre o instante em que a inundação atinge o alvo (T_0, primeira amostra com evento de inundação no sensor) e o instante em que o controlador instala a regra de bloqueio da origem (T_1, registro de bloqueio no log do DDSO). A sincronização dos relógios entre as máquinas, mantida por NTP, assegura a comparabilidade das marcas temporais.

4.2.3 Resultados de Eficácia

A Tabela 4.2 consolida a eficácia das quatro condições, com as médias de tempo de exposição do alvo e os respectivos intervalos de confiança de 95%, estimados pela distribuição t de Student. O tamanho amostral de cada condição consta na tabela, após a exclusão das capturas que não satisfizeram os critérios de validade descritos no Apêndice D, a saber, a confirmação de que o ataque atingiu o alvo e a ausência de sobreposição de janelas de disparo.

Tabela 4.2: Síntese da eficácia das estratégias de ancoragem sob SYN *flood* de origem fixa. Intervalos de confiança de 95% pela distribuição t de Student. O tamanho amostral n por condição consta na coluna respectiva.

Estratégia	**Âncora**	n	**Exposição (s)**	**IC 95%**	**Redução**
off (referência)	—	26	24,50	[21,76; 27,24]	—
ip	`IPV4_SRC`	30	2,00	[1,83; 2,17]	91,8%
mac	`ETH_SRC`	30	2,77	[2,03; 3,50]	88,7%
auto	`IPV4_SRC`	29	2,62	[1,76; 3,48]	89,3%

Na condição de referência, o alvo permaneceu sob inundação por 24,50 segundos em média, com pico de taxa de pacotes próximo de cinquenta mil pacotes por segundo. Esse valor, embora corresponda a uma janela inferior à duração nominal do disparo, é coerente com a contenção parcial exercida pelo *firewall* de borda sobre origens estáveis, fenômeno que se repete no vetor UDP e que é discutido na Seção 4.7. As três estratégias de mitigação reduziram a exposição para a faixa de poucos segundos, com eficácia entre 88,7% e 91,8%.

A estratégia *ip* reduziu a exposição para 2,00 segundos em média, com intervalo de confiança entre 1,83 e 2,17 segundos, uma redução de 91,8%. Como a origem do ataque é real e estável, a regra ancorada em `IPV4_SRC` casa os pacotes do agressor e os descarta no plano de dados já no primeiro ciclo de atuação. As estratégias *mac* e *auto* alcançaram exposições próximas, de 2,77 e 2,62 segundos, com reduções de 88,7% e 89,3%. O registro forense da estratégia *auto* confirma que, diante de origem confiável, o orquestrador permaneceu na âncora de camada de rede e não acionou a escalada para o endereço físico, comportamento idêntico ao observado nos vetores UDP e GRE. A diferença em relação à condição de referência é estatisticamente significativa: o teste de Mann-Whitney U (bicaudal) indicou $p < 0{,}001$ e tamanho de efeito máximo (delta de Cliff $\delta = +1{,}00$) para as três estratégias, conforme a análise estatística complementar da Seção 4.8 (Tabela 4.11). A Figura 4.1 contrasta a persistência do ataque na condição de referência com a contenção obtida pela estratégia *auto*. As curvas reproduzem a telemetria do alvo amostra a amostra, sem suavização, na repetição de exposição mediana de cada condição. Sob a estratégia *auto*, o bloqueio por `IPV4_SRC` contém o ataque em poucos segundos. O eixo do tempo é apresentado até 75 segundos para realçar o contraste; a série completa registra o retorno do tráfego ao patamar de fundo após a remoção da regra de descarte (`UNBLOCK`).

O pico de taxa de pacotes sob mitigação permanece em patamar comparável ao da condição de referência. Esse comportamento é esperado e não contradiz a eficácia da defesa. O pico ocorre no instante inicial do ataque, antes da atuação do controlador, e reflete a intensidade bruta da inundação ao alcançar a vítima. O que distingue as duas condições não é a magnitude instantânea do tráfego, mas a sua persistência. Sob mitigação, a inundação é interrompida poucos segundos após o pico, enquanto na referência ela se mantém por toda a janela observada.

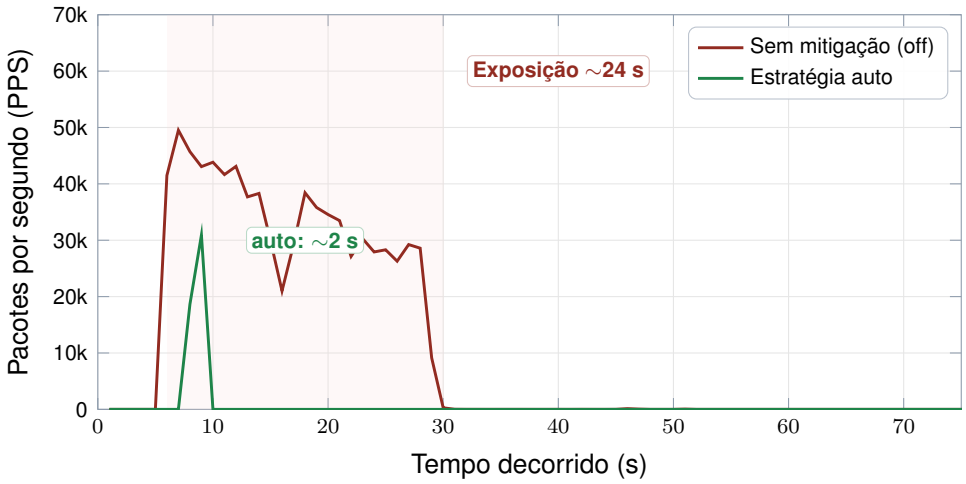

Figura 4.1: Taxa de pacotes por segundo na interface da vítima durante o SYN *flood* de origem fixa.

4.2.4 Latência de Mitigação

A latência de mitigação foi medida na estratégia *ip*, que ancora o descarte no endereço IP de origem e constitui a referência direta de tempo de reação para o vetor de origem fixa. A latência média foi de 0,71 segundo, com desvio padrão de 0,30 segundo e intervalo de confiança de 95% de $[0{,}59;\ 0{,}83]$ segundo, estimado sobre vinte e sete repetições nas quais o registro de bloqueio pôde ser pareado de forma inequívoca ao início da inundação no sensor. Os valores individuais distribuíram-se entre 0,22 e 1,17 segundos. A Tabela 4.3 consolida essas medições.

Tabela 4.3: Latência de mitigação do DDSO sob o vetor SYN *flood* na estratégia *ip*, com período de varredura de um segundo.

Indicador	**Valor**
Amostras pareadas	27
Latência média de mitigação	0,71 s
Desvio padrão	0,30 s
Intervalo de confiança de 95%	$[0{,}59;\ 0{,}83]$ s
Faixa observada	0,22 a 1,17 s
Equivalência ao ciclo de varredura (1 s)	cerca de meio ciclo

A dispersão observada na latência tem explicação direta no mecanismo de detecção. O DDSO inspeciona os contadores de fluxo do comutador em intervalos regulares de um segundo. Como o instante de chegada da inundação à vítima não guarda relação de fase com o ciclo de varredura do controlador, o atraso entre o início do ataque e a varredura seguinte distribui-se, em primeira aproximação, de forma uniforme no intervalo de zero a um segundo. A média empírica de 0,71 segundo, somada à latência de instalação da regra pela interface REST, da ordem de poucos milissegundos, aproxima-se da metade do período de varredura, valor coerente com o comportamento de um amostrador periódico. A comparação com a campanha conduzida sob período de varredura de dois segundos, na qual a latência média se situou próxima de um segundo,

confirma que a latência total escala com o período de varredura, e não com a operação de instalação da regra.

Essa decomposição evidencia que a latência total do sistema é dominada pela etapa de observação, e não pela etapa de ação. O tempo necessário para que o controlador instale efetivamente a regra de bloqueio no comutador, medido a partir das requisições à interface REST do ONOS, manteve-se na faixa de poucos milissegundos em todas as repetições, valor desprezível diante da granularidade da varredura. A redução da latência de mitigação neste sistema passa, portanto, pela redução do período de varredura, e não pela otimização da operação de bloqueio, que já opera em escala de milissegundos. Essa latência mede o intervalo de reação do plano de controle, do início da inundação até a emissão da regra de bloqueio registrada pelo orquestrador, e não o instante da cessação física do tráfego no comutador, que é capturado de forma independente pela métrica de tempo de exposição do alvo. Como a coleta opera em varredura de um segundo, os valores médios inferiores a esse intervalo carregam uma incerteza da ordem de um ciclo de amostragem, o que recomenda cautela na sua leitura absoluta.

4.3 FENÔMENOS DE ASSINCRONIA DE HARDWARE: A GÊNESE DAS REGRAS ÓRFÃS

A avaliação da eficácia, embora favorável, conduziu à descoberta de um limite prático da teoria SDN quando confrontada com hardware comercial sob estresse. Com o módulo DDSO em modo de prevenção, o sistema detectava o desvio de tráfego e iniciava a injeção de regras de descarte por mensagens `FLOW_MOD`. Em ataques de intensidade moderada, o ciclo de mitigação funcionou conforme o esperado. Sob estresse volumétrico severo, porém, o canal OpenFlow manifestou uma degradação na integridade das mensagens de remoção de fluxo.

4.3.1 O Comportamento Fantasma na Memória TCAM

A política de defesa de borda exige que, uma vez cessado o tráfego agressor, a regra de bloqueio seja removida, tanto para liberar espaço na escassa memória TCAM quanto para não penalizar tráfego legítimo em cenários de endereços compartilhados ou NAT. Ao concluir seus ciclos de normalização, o DDSO emitiu a instrução de remoção para o comutador. Foi nesse ponto da pesquisa empírica que se registrou o fenômeno da assincronia.

A interface REST do ONOS confirmou a remoção lógica com status HTTP 204. No estado interno do controlador, a regra deixou de existir. No plano de dados, entretanto, a regra permaneceu instalada na tabela de fluxos do Dell S3048-ON, continuando a descartar tráfego de forma indefinida, sem que o controlador tivesse ciência de sua persistência. Esse descompasso entre a remoção lógica no controlador e a remoção física no comutador é consistente com o comportamento assíncrono das operações de escrita e exclusão de regras em *hardware* documentado por Wijeratne *et al.* [9], e configura o fenômeno aqui denominado regra órfã. Em um simulador como o Mininet,

no qual a memória é volátil e atua de modo quase instantâneo, esse erro permaneceria invisível e produziria métricas de sucesso que não se sustentam em hardware real.

4.4 AVALIAÇÃO DO DDSO E MITIGAÇÃO EM MALHA FECHADA

Para sanar a vulnerabilidade descoberta na seção anterior, avaliou-se o desempenho do mecanismo de verificação de consistência em malha fechada (*State Checker*) embutido no DDSO. A abordagem reintroduziu a auditoria proativa. O orquestrador deixou de confiar no sucesso lógico reportado pelo ONOS e passou a consultar de forma ativa os contadores de hardware do comutador.

4.4.1 Mecanismo de Reação e Consistência

O fenômeno das regras órfãs foi observado nas versões iniciais do orquestrador, que confiavam na confirmação lógica (HTTP 204) reportada pelo controlador. A evolução do DDSO incorporou a verificação de consistência em malha fechada: o orquestrador passou a auditar continuamente os contadores de fluxo do comutador e, ao identificar que uma regra supostamente removida persistia incrementando seus contadores de *bytes* e pacotes no plano de dados, despachou comandos de remoção iterativos, forçando a liberação da entrada correspondente no plano de dados. Com esse mecanismo ativo, a consistência entre o estado lógico do controlador e o estado físico do comutador foi restabelecida ao final de cada evento de mitigação avaliado. A Tabela 4.4 consolida os indicadores de desempenho do orquestrador, a partir das repetições do vetor SYN.

A observação de estado empregada pelo verificador merece precisão metodológica, dado que a reconciliação entre planos envolve estados distintos. Distinguem-se quatro: o estado desejado pela aplicação de segurança, o estado lógico registrado pelo controlador, o estado reportado pelo agente OpenFlow do comutador e o estado efetivamente instalado no *hardware* de encaminhamento. O *State Checker* não inspeciona diretamente a memória do equipamento; ele consulta os contadores de fluxo expostos pela *Northbound* API do ONOS (estatísticas por fluxo, entre elas o tempo de vida da entrada e os contadores de pacotes e *bytes*), que refletem o estado reportado pelo agente OpenFlow do comutador. A regra órfã manifesta-se, nessa observação, quando uma entrada que o plano de controle considera removida continua a ser reportada pelo agente do equipamento com contadores em incremento. O mecanismo atua, portanto, sobre a divergência entre o estado lógico do controlador e o estado de encaminhamento reportado pelo equipamento, e não sobre uma leitura direta do silício, o que delimita com precisão o seu alcance.

A auditoria dinâmica mostrou-se o fator determinante para a viabilidade operacional do paradigma SDN em bordas de rede sob ataque. Ao deixar de confiar exclusivamente na confirmação lógica do controlador e passar a verificar o estado real do plano de dados, o orquestrador converteu uma falha silenciosa, invisível em ambientes emulados, em uma condição detectável e corrigível em *hardware* comercial.

Tabela 4.4: Indicadores de desempenho do orquestrador DDSO sob estresse volumétrico.

Indicador de Desempenho	**Valor**	**Observação**
Latência de mitigação ($T_1 - T_0$)	0,71 s	Dominada pelo ciclo de varredura (*polling*) de 1 s.
Latência de injeção de regra (*Drop*)	poucos ms	Tempo da requisição REST até a confirmação pelo ONOS.
Tempo de exposição do alvo (com mitigação)	2,00 s	Redução de 91,8% em relação à referência (estratégia *ip*).
Regras órfãs (versões sem auditoria)	observadas	Persistência de regras na TCAM após remoção lógica.
Consistência (com *State Checker*)	restabelecida	Reconciliação do estado ao final de cada mitigação.

4.5 MITIGAÇÃO SOB FALSIFICAÇÃO DE ENDEREÇO DE ORIGEM (*IP SPOOFING*)

As seções anteriores submeteram a defesa a inundações cuja origem mantinha um endereço IP fixo e legítimo. A presente seção avalia uma condição mais adversa, na qual o atacante falsifica o endereço de origem a cada pacote. A técnica é comum em ataques volumétricos porque dificulta a atribuição do tráfego e inviabiliza a filtragem por endereço de origem [51]. O que se pretende observar é como o orquestrador se comporta quando o identificador de camada de rede deixa de ser estável, e se ainda resta, na borda, alguma âncora de identidade que possa ser explorada para conter o ataque. A auditoria por contador descrita na seção anterior, conforme se verá, é a peça que torna essa contenção possível.

4.5.1 Cenário, Alvo e Endereçamento Observado

O ataque partiu dos mesmos três dispositivos infectados da VLAN 22 (164.72.22.1, 164.72.22.2 e 164.72.22.3), conectados à porta 9 do comutador, e teve como destino o hospedeiro `vm-target-h-01` (164.72.55.1), na VLAN 55. O comando emitido pelo servidor de comando e controle acionou a falsificação de origem. Na linhagem Mirai utilizada, o valor 255.255.255.255 passado ao parâmetro de origem aciona a randomização do endereço a cada pacote, comportamento confirmado pela própria ajuda do *bot*. No perímetro, o efeito foi imediato. O *firewall* passou a registrar um grande número de endereços de origem diferentes, todos endereçados ao mesmo alvo.

Um aspecto do endereçamento exibido pelo painel do orquestrador precisa ser esclarecido para que as tabelas não sejam lidas de forma ambígua. O alvo do ataque é, sem dúvida, o hospedeiro 164.72.55.1, que é a informação contabilizada pelo *firewall* de borda, o qual opera na camada de rede e enxerga o endereço IP de destino real. O painel Alvos Sob Ataque do DDSO, por outro lado, reporta o destino como o gateway de borda, alternando entre endereços terminados em .254. A consulta à base de hospedeiros do ONOS resolve o aparente conflito. Todos esses endereços

pertencem a um único endereço físico (BC:24:11:8B:48:F4), o roteador e *firewall* que interliga as VLANs por subinterfaces, uma para cada VLAN. Como os *bots* e o alvo residem em VLANs distintas, o tráfego é roteado, e o quadro Ethernet emitido pelos *bots* carrega, na camada de enlace, o endereço físico do gateway como destino, e não o do hospedeiro final. O orquestrador, que observa e agrega o plano de dados na camada de enlace, informa esse endereço de gateway como destino. O mesmo tráfego acaba descrito em dois níveis, com o IP de destino real na camada de rede e o próximo salto na camada de enlace. Essa propriedade volta a aparecer mais adiante, pois é justamente ela que explica o resultado central da seção.

4.5.2 Estratégias de Ancoragem da Regra de Descarte

A regra de descarte instalada via OpenFlow pode ancorar a correspondência em dois critérios do cabeçalho, o endereço IP de origem (`IPV4_SRC`) ou o endereço físico de origem (`ETH_SRC`). O orquestrador foi instrumentado com três estratégias de ancoragem, selecionáveis por parâmetro de execução, que constituem o desenho experimental desta avaliação. Elas não devem ser entendidas como três opções concorrentes para uso operacional. Foram concebidas para isolar variáveis e tornar o resultado demonstrável.

A estratégia *ip* ancora o descarte apenas em `IPV4_SRC` e reproduz o comportamento das versões anteriores do orquestrador, equivalente à abordagem convencional de bloqueio por endereço de origem. Cumpre o papel de tratamento de controle e expõe o problema sob falsificação. A estratégia *mac* ancora o descarte em `ETH_SRC` desde o primeiro instante e serve para medir, de forma isolada, a eficácia da âncora de camada de enlace, sem qualquer lógica intermediária. A estratégia *auto*, que é o padrão operacional e a contribuição proposta neste trabalho, inicia o bloqueio por `IPV4_SRC` e, no ciclo de verificação de estado, passa a observar o contador da própria regra de descarte. Quando esse contador permanece inerte enquanto o *flood* persiste nas estatísticas agregadas, o orquestrador infere que o endereço IP de origem é volátil, escala a regra para `ETH_SRC` e reclassifica o evento como `FLOOD/SPOOFED`.

4.5.3 Resultados e Análise

Cada uma das quatro condições foi exercitada em trinta repetições, sob o mesmo *flood* ACK com origem forjada, de vazão agregada próxima de cinquenta mil pacotes por segundo na entrada da vítima. A falsificação foi acionada pelo parâmetro de origem da linhagem Mirai, que randomiza o endereço IP a cada pacote. A condição de referência sem mitigação, identificada por *off*, fornece a base de comparação. A telemetria foi capturada na interface `enp6s19` do alvo, com amostragem a cada segundo, e o controlador operou com período de varredura de um segundo, em regime idêntico ao dos vetores SYN, GRE e UDP, o que torna esta avaliação diretamente comparável às demais do capítulo. As latências de instalação da regra pela interface REST permaneceram na ordem de poucos milissegundos em todas as condições.

A Tabela 4.5 apresenta o tempo de exposição do alvo em cada repetição, nas quatro condições.

A Tabela 4.6 consolida as médias, os intervalos de confiança de 95%, estimados pela distribuição t de Student, e a redução de exposição em relação à referência. O tamanho amostral de cada condição consta na tabela de síntese, após a exclusão das capturas que não satisfizeram os critérios de validade descritos no Apêndice D, a saber, a confirmação de que o ataque atingiu o alvo e a ausência de sobreposição de janelas de disparo na condição de referência.

Tabela 4.5: Tempo de exposição do alvo ao ACK *flood* com origem forjada, por repetição, nas quatro condições avaliadas, sob varredura de um segundo (segundos).

Rep.	off	ip	mac	auto	Rep.	off	ip	mac	auto
1	57	60	1	4	16	56	55	1	4
2	60	55	2	4	17	60	56	2	4
3	60	47	2	4	18	59	59	2	4
4	60	50	1	4	19	54	58	1	4
5	58	58	2	4	20	57	58	2	5
6	60	58	2	4	21	59	49	1	4
7	60	59	3	5	22	56	49	2	4
8	60	58	2	4	23	50	56	2	4
9	60	58	2	4	24	50	55	2	5
10	59	60	2	4	25	58	60	2	5
11	60	55	2	4	26	55	56	2	3
12	58	58	2	4	27	57	58	2	5
13	32	56	2	4	28	59	56	2	3
14	58	56	2	5	29	57	59	2	5
15	59	56	2	4	30	55	58	2	4

Tabela 4.6: Síntese da eficácia das estratégias de ancoragem sob ACK *flood* com origem forjada, sob varredura de um segundo. Intervalos de confiança de 95% pela distribuição t de Student.

Estratégia	Âncora	n	Exposição (s)	IC 95%	Redução
off (referência)	—	30	56,77	[54,75; 58,78]	—
ip (controle)	`IPV4_SRC`	30	56,20	[54,94; 57,46]	1,0%
mac	`ETH_SRC`	30	1,87	[1,70; 2,03]	96,7%
auto	`IPV4_SRC` → `ETH_SRC`	30	4,17	[3,97; 4,36]	92,7%

4.5.3.1 Condição de Referência e Falha do Bloqueio por Endereço de Rede

Na condição de referência, o alvo permaneceu sob inundação por 56,77 segundos em média, com intervalo de confiança de 95% entre 54,75 e 58,78 segundos. O valor corresponde, na prática, à duração do disparo, e confirma que, sem mitigação, nada interrompe o *flood* antes que o agressor encerre o ataque. O número contrasta com o vetor SYN de origem fixa, em que a exposição de referência ficou em torno de vinte e quatro segundos. A diferença tem causa identificável na falsificação de origem. Ao espalhar o tráfego por endereços de rede que mudam a cada pacote, o ataque frustra a filtragem por estado no perímetro e prolonga a exposição até o fim do disparo. Na condição de referência, em que nenhuma regra de bloqueio é instalada, o tempo de exposição

corresponde à duração efetiva do disparo, e a variação entre repetições reflete o comportamento do próprio gerador de ataque, e não qualquer ação de defesa.

A estratégia *ip* reproduziu a condição de referência. A exposição média foi de 56,20 segundos, com intervalo de confiança entre 54,94 e 57,46 segundos. O teste de Mann-Whitney U não detectou diferença estatisticamente significativa entre a estratégia *ip* e a condição *off* ($U = 568$; $p = 0{,}079$), e o tamanho de efeito foi pequeno (delta de Cliff $\delta = +0{,}26$), com redução de apenas 1,0%. O resultado caracteriza a ausência de eficácia prática do bloqueio por endereço IP sob origem forjada, e não uma estratégia apenas menos eficiente, conforme a análise estatística complementar da Seção 4.8 (Tabela 4.11). O registro do orquestrador mostra a causa. O detector identificou o ataque e instalou as regras de descarte ancoradas em `IPV4_SRC` sobre os endereços observados, com confirmação REST bem-sucedida. Os contadores de descarte dessas regras, contudo, permaneceram inertes, na casa de poucas dezenas de pacotes, ao mesmo tempo em que o painel de alvos seguia acusando ataque ativo. A regra ancorada em um endereço IP não casa os pacotes do *flood*, cuja origem é forjada e variável, e o tráfego atravessa o comutador sem ser descartado no plano de dados. O registro forense ainda exibe um ciclo que se repete, de bloqueio, normalização aparente e novo bloqueio, sinal de que o controlador tratava como bem-sucedida uma ação sem efeito no plano de dados. Esse é o desfecho esperado para o tratamento de controle, e é o que evidencia a falha do bloqueio por endereço de origem diante da falsificação. O resultado tem valor metodológico, pois isola a variável de interesse. Como demonstrado no vetor UDP de origem fixa, a mesma estratégia *ip* reduz a exposição para cerca de dois segundos quando a origem é estável. A falha registrada aqui não decorre, portanto, de uma deficiência intrínseca do critério de correspondência, mas exclusivamente da volatilidade do endereço de origem.

4.5.3.2 Eficácia das Âncoras Resistentes à Falsificação

A estratégia *mac* reduziu a exposição para 1,87 segundos em média, com intervalo de confiança entre 1,70 e 2,03 segundos, uma redução de 96,7% em relação à referência. A regra ancorada em `ETH_SRC` interrompe o *flood* já no primeiro ciclo de atuação. O registro forense contabiliza os bloqueios diretos por endereço físico sobre os três dispositivos, sem nenhum escalonamento. O endereço físico de origem é estável porque o quadro Ethernet parte da interface real do dispositivo, ainda que o endereço IP venha forjado, e por isso a regra por `ETH_SRC` casa todos os pacotes daquele hospedeiro na borda de enlace. A eficácia da âncora física confirma-se de forma independente da família de protocolo e da volatilidade do endereço de rede.

A estratégia *auto* reduziu a exposição para 4,17 segundos em média, com intervalo de confiança entre 3,97 e 4,36 segundos, uma redução de 92,7%. O orquestrador começou pelo bloqueio em `IPV4_SRC` e, ao constatar que o contador da própria regra seguia inerte enquanto o *flood* persistia nas estatísticas agregadas, concluiu que a origem era volátil, escalou a regra para `ETH_SRC` e reclassificou o evento como `FLOOD/SPOOFED`. As noventa e três escaladas registradas ao longo das trinta repetições ancoraram, sem exceção, nos três endereços físicos reais dos dispositivos atacantes (`BC:24:11:34:04:01`, `BC:24:11:CA:DA:02` e `BC:24:11:77:62:03`), e ne-

nhuma recaiu sobre o endereço do gateway ou do próprio alvo. O total de noventa e três supera as noventa escaladas nominais (três origens em trinta repetições) por conta de re-escaladas pontuais, em que uma origem cujo bloqueio por IP foi removido voltou a cruzar o limiar e exigiu nova migração de âncora dentro da mesma repetição. A Tabela 4.7 quantifica a transição.

Tabela 4.7: Latência de escalonamento da regra de `IPV4_SRC` para `ETH_SRC` na estratégia *auto*, medida no registro forense do orquestrador, sob varredura de um segundo.

Indicador	**Valor**
Escaladas registradas (3 origens, 30 repetições)	93
Latência média de escalonamento	2,13 s
Desvio padrão	0,03 s
Intervalo de confiança de 95%	[2,13; 2,14] s
Faixa observada	2,10 a 2,21 s
Equivalência em ciclos de varredura (1 s)	cerca de 2 ciclos

4.5.3.3 Decomposição da Latência por Ciclos de Observação

O escalonamento ocorreu com regularidade marcante, em 2,13 segundos em média, com desvio padrão de apenas 0,03 segundo. A baixa dispersão indica que a transição não é fortuita, e sim governada por um número inteiro de ciclos de varredura. Sob o período de um segundo, o valor corresponde a cerca de dois ciclos, e essa decomposição esclarece o mecanismo de forma mais nítida do que a campanha anterior, em que o período de dois segundos comprimia os dois ciclos em uma única janela e obscurecia a contagem.

A escalada exige, por construção, dois ciclos de observação. No primeiro, o detector identifica o *flood*, instala a regra ancorada em `IPV4_SRC` e registra o bloqueio. No segundo, audita o contador daquela regra, constata que permanece inerte enquanto o ataque persiste, infere a volatilidade da origem e escala para `ETH_SRC`. O passo intermediário é o ciclo de evidência previsto no projeto do orquestrador, que exige a confirmação da ineficácia antes de migrar de âncora, de modo a evitar escalonamentos precipitados sobre origens legítimas. A latência de 2,13 segundos é, portanto, a soma de dois ciclos de detecção sequenciais, acrescida das latências de instalação de regra, da ordem de milissegundos. A relação entre as duas estratégias eficazes corrobora a leitura. A diferença entre as exposições médias da *auto* e da *mac*, de 4,17 menos 1,87, resulta em 2,30 segundos, valor próximo da latência de escalonamento de 2,13 segundos medida de forma independente no registro do controlador. A estratégia *auto* equivale, então, à *mac* acrescida do custo dos ciclos gastos em tentar o bloqueio por IP e verificar a sua ineficácia antes de migrar para o endereço físico. Duas vias de medição que não se comunicam, a telemetria do alvo e o registro forense, convergem para o mesmo valor, e essa coincidência sustenta a leitura proposta.

A comparação com a campanha conduzida sob varredura de dois segundos confirma a tese do amostrador periódico. Naquele regime, a escalada ocorria em torno de dois segundos, valor que

aparentava corresponder a um único ciclo. Sob o período de um segundo, a mesma transição ocorre em torno de dois segundos, o que revela que se trata, na realidade, de dois ciclos de observação, e não de um. O tempo total da escalada escala com o período de varredura na proporção de dois ciclos, e não com a operação de instalação da regra, que permanece em escala de milissegundos. A latência da defesa é, assim, dominada pela etapa de observação, em coerência com o que foi estabelecido na análise do vetor SYN.

4.5.3.4 Contraste de Persistência

A Figura 4.2 contrasta a persistência do ataque sem mitigação com a contenção obtida pela estratégia *auto*, em repetições cujas exposições coincidem com as medianas das trinta. As séries são extraídas diretamente da telemetria da vítima, amostra a amostra, sem suavização, o que preserva a variação natural da taxa de pacotes característica de uma inundação real. Sem mitigação, os endereços de origem forjados atravessam a filtragem por estado do perímetro e a vítima permanece sob inundação por toda a janela do disparo. Sob a estratégia *auto*, o bloqueio inicial por `IPV4_SRC` se mostra inerte e, após dois ciclos de observação, o orquestrador escala para `ETH_SRC`, levando a taxa no alvo a zero. O eixo do tempo é apresentado até 75 segundos; a série completa registra o retorno ao patamar de fundo após o `UNBLOCK`.

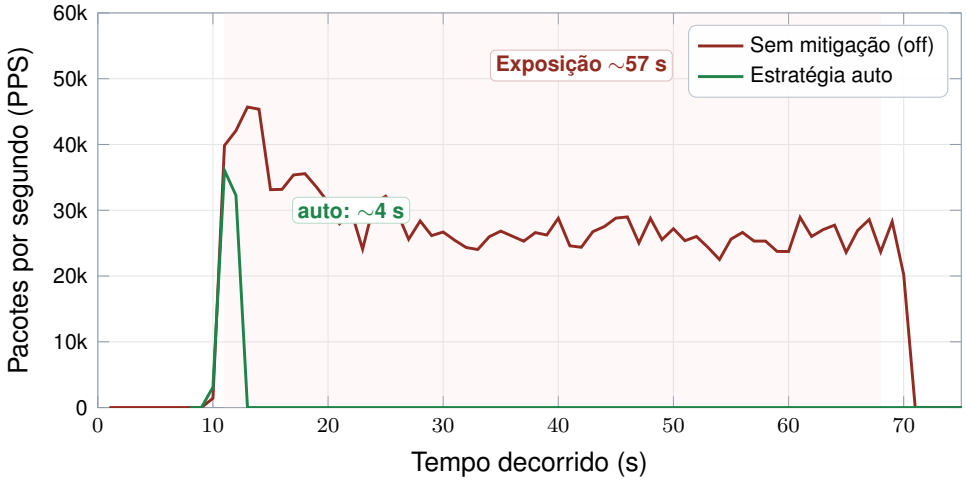

Figura 4.2: Taxa de pacotes por segundo na interface da vítima durante o ACK *flood* com origem forjada.

Os picos de taxa de pacotes na vítima mantiveram patamar semelhante entre as condições. Eles ocorrem no instante inicial do ataque, antes de qualquer atuação do controlador, e refletem a intensidade bruta da inundação ao alcançar a vítima. O que separa as estratégias eficazes das demais não é a magnitude instantânea do tráfego, mas a brevidade da janela em que ele se sustenta. O mérito da estratégia *auto* não está na velocidade, em que a *mac* a supera por um ciclo, e sim na autonomia. Ela dispensa o conhecimento prévio de que o ataque é falsificado e infere essa condição em tempo de execução, a partir do comportamento observável dos contadores de *hardware*.

4.5.4 A Âncora de Identidade na Camada de Enlace

A inspeção dos fluxos OpenFlow durante o ataque fornece a evidência que fundamenta a estratégia adotada. Embora o *firewall* registrasse uma grande quantidade de endereços IP de origem distintos, o comutador encaminhava o ataque por um número reduzido e fixo de fluxos, um por dispositivo, identificados pelo par formado pelo endereço físico de origem e o de destino. Os três endereços físicos de origem observados nesses fluxos permaneceram constantes durante todo o ataque (BC:24:11:CA:DA:02, BC:24:11:77:62:03 e BC:24:11:34:04:01), em contraste com a quantidade elevada de endereços IP forjados.

A assimetria decorre do modo de operação do controlador. A aplicação de encaminhamento reativo do ONOS agrega o tráfego pelo endereço físico, e não pelo endereço IP. Em consequência, os endereços IP forjados sequer se tornam visíveis ao controlador, pois ficam ocultos sob a agregação de camada de enlace. É o mesmo fenômeno descrito na Subseção 4.5.1 a respeito do destino. Como o orquestrador raciocina na camada de enlace, o endereço de camada de rede, forjado na origem ou pertencente ao gateway no destino, não participa da agregação. A âncora de identidade estável e observável no plano de dados é, portanto, o endereço físico de origem, o que justifica em termos técnicos o bloqueio por `ETH_SRC` como vetor de mitigação resistente à falsificação.

O protocolo OpenFlow [2] dispõe de campos de correspondência na camada de transporte, como a porta de destino (`TCP_DST` e `UDP_DST`), o que, em princípio, permitiria refinar a regra de descarte para atingir apenas o serviço sob ataque, preservando os demais fluxos do hospedeiro. Essa seletividade, contudo, não altera o desfecho sob falsificação de origem. O problema observado na estratégia *ip* não decorre da granularidade da regra, mas da ausência de correspondência: uma regra ancorada em um `IPV4_SRC` volátil não casa pacote algum, e acrescentar-lhe um critério de porta apenas restringiria ainda mais um filtro que já não atua. A âncora eficaz permanece sendo o endereço físico de origem, observável e estável na borda de enlace. O refinamento por porta de destino tem pertinência no caso de origem fixa sobre hospedeiros legítimos comprometidos, cenário em que a granularidade evitaria a negação de serviço sobre os serviços benignos, e fica registrado como desdobramento natural para trabalhos futuros.

Esse resultado tem duas condições de contorno que delimitam o seu alcance. A primeira é a dependência do domínio de camada de enlace. O bloqueio por `ETH_SRC` só é eficaz enquanto o endereço físico real do dispositivo é observável no ponto em que o controlador atua. No ambiente de testes, os *bots* residem na mesma VLAN e o comutador observa o endereço físico real na porta de entrada. Caso o tráfego fosse roteado por um salto de camada de rede antes desse ponto, o endereço físico de origem passaria a ser o do roteador e a distinção entre os dispositivos se perderia. Trata-se de uma propriedade do posicionamento de borda da defesa, e não de uma solução universal. A segunda condição diz respeito à granularidade. A regra por `ETH_SRC` descarta todo o tráfego do hospedeiro de origem, e não apenas o tráfego de ataque. Para um *bot* dedicado, esse comportamento é aceitável e até desejável. Em um hospedeiro legítimo comprometido, equivaleria a uma negação de serviço sobre os seus serviços benignos.

4.5.5 Nomenclatura dos Vetores de Ataque

Os rótulos exibidos pelo orquestrador descrevem a técnica de ataque observada, e não a propriedade de falsificação, que é ortogonal e foi inferida pelo mecanismo de escalonamento. A Tabela 4.8 associa cada rótulo ao nome técnico do vetor, ao vetor correspondente na família Mirai e à propriedade de origem. O rótulo `FLOOD` é agnóstico em relação à *flag* TCP e abrange tanto o SYN *flood* quanto o ACK *flood*. O sufixo `/SPOOFED` só é acrescentado quando há evidência de origem forjada.

Tabela 4.8: Correspondência entre os rótulos do orquestrador, os nomes técnicos dos vetores e os vetores da família Mirai.

Rótulo DDSO	**Vetor (nome técnico)**	**Vetor Mirai**	**Origem**
`FLOOD`	TCP flood (SYN ou ACK)	`ATK_VEC_SYN` `ATK_VEC_ACK`	Fixa ou forjada
`FLOOD/ SPOOFED`	TCP flood com falsificação	`ATK_VEC_SYN` `ATK_VEC_ACK`	Forjada
`GRE-FLOOD`	GRE flood	`ATK_VEC_GREIP` `ATK_VEC_GREETH`	Fixa ou forjada
`DNS- TORTURE`	DNS Water Torture	`ATK_VEC_DNS`	Subdomínio aleatório
`UDP-AMP`	Amplificação e reflexão UDP	`ATK_VEC_UDP` (variantes)	Refletida
`SCAN`	Varredura ICMP e de portas	Reconhecimento (`scanner.c`)	Fixa
`BRUTE`	Força bruta de credenciais	Propagação (`scanner.c`, *loader*)	Fixa

4.6 MITIGAÇÃO SOB INUNDAÇÃO GRE DISTRIBUÍDA DE ORIGEM REAL

As três seções anteriores percorreram inundações de transporte, com o vetor SYN de origem fixa, o vetor ACK sob falsificação de origem e o vetor UDP de origem estável. A presente seção avalia o vetor GRE, que acrescenta ao quadro experimental um protocolo de encapsulamento distinto, identificado pelo número 47 no cabeçalho de camada de rede, e um regime de disparo genuinamente distribuído, em que os três dispositivos infectados atacam de forma simultânea a partir de seus endereços reais. O vetor cumpre dois papéis na argumentação. Estende a avaliação a uma família de protocolo que não havia sido coberta pelas inundações de transporte, e fornece o caso de origem confiável sob ataque distribuído, que completa o contraste com o ACK falsificado e reforça a leitura proposta na seção anterior, segundo a qual é a falsificação de origem, e não o protocolo em si, que determina a âncora necessária para a contenção.

4.6.1 Cenário e Caracterização do Vetor

O ataque partiu dos mesmos três dispositivos infectados da VLAN 22 (164.72.22.1, 164.72.22.2 e 164.72.22.3) e teve como destino o hospedeiro `vm-target-h-01` (164.72.55.1), na VLAN 55. O disparo encapsulou o tráfego em pacotes do protocolo GRE, correspondente aos vetores `ATK_VEC_GREIP` e `ATK_VEC_GREETH` da família Mirai, cuja taxonomia consta na Tabela 4.8. Na versão do orquestrador empregada nestes ensaios, a inundação é detectada pela elevação volumétrica e registrada sob o evento `FLOOD`, sem rotulação específica de protocolo no plano de controle. Diferentemente do ACK, a inundação GRE foi conduzida a partir dos endereços reais dos dispositivos, sem falsificação dinâmica de origem. Essa condição aproxima o vetor do regime já observado nos vetores SYN e UDP, em que o endereço IP de origem é estável e, portanto, casável por uma regra ancorada em `IPV4_SRC`.

Um traço próprio deste ensaio, ausente nos demais vetores, é o caráter distribuído da inundação. Enquanto nos ensaios anteriores a contenção atuava sobre origens que se tornavam ativas de forma praticamente simultânea, aqui os três dispositivos sustentaram o ataque em paralelo durante toda a janela, o que permite observar como a defesa se comporta quando a mitigação de cada origem se completa de forma independente. Esse aspecto, discutido na análise dos resultados, explica a dispersão observada na estratégia adaptativa e constitui uma evidência adicional do funcionamento por origem do mecanismo de descarte.

A telemetria foi capturada na interface `enp6s19` da vítima, com amostragem a cada segundo, e a janela de coleta de cada repetição foi estendida para cerca de cento e quinze segundos, de modo a abranger com folga o disparo e o retorno à normalidade. Na condição de referência (*off*), o módulo DDSO operou inativo, sem instalar qualquer regra de bloqueio durante o ataque. A ausência de mitigação é evidenciada pela própria telemetria do alvo, que registrou inundação pela duração integral do disparo em todas as repetições, com tempo de exposição de 60,73 segundos e desvio padrão de apenas 0,50 segundo, o que descarta qualquer interrupção por ação de defesa nessa condição. O registro forense do orquestrador foi coletado em paralelo nas condições de mitigação.

4.6.2 Procedimento e Estratégias Avaliadas

O desenho experimental reproduz a estrutura de quatro condições adotada no vetor ACK, com a condição de referência sem mitigação (*off*) e as três estratégias de ancoragem (*ip*, *mac* e *auto*) descritas na Subseção 4.5.2. A repetição das três estratégias, mesmo em um vetor de origem estável em que a estratégia *ip* já é suficiente, tem propósito metodológico. Permite verificar se a estratégia adaptativa, diante de origem confiável, de fato permanece na âncora de camada de rede sem escalar para o endereço físico, isto é, se o comportamento da *auto* é coerente com o observado no vetor UDP, no qual a escalada jamais foi acionada. Cada condição foi repetida trinta vezes, em coerência com o tamanho amostral adotado nos demais vetores do capítulo, o que permite caracterizar com nitidez a cauda de dispersão da estratégia adaptativa.

Os ensaios do vetor GRE foram conduzidos com o orquestrador DDSO na versão final v19.3.1, a mesma empregada em todos os vetores reportados neste capítulo. A unicidade da versão entre as campanhas assegura a rastreabilidade e a comparabilidade dos resultados: o caminho de detecção e de instalação da regra é idêntico entre os vetores, e o ciclo de varredura, que domina a latência de mitigação, manteve-se constante em um segundo ao longo de toda a avaliação.

4.6.3 Resultados e Análise

A Tabela 4.9 consolida a eficácia das quatro condições, com as médias de tempo de exposição do alvo e os respectivos intervalos de confiança de 95%, estimados pela distribuição t de Student com vinte e nove graus de liberdade. Na condição de referência, o alvo permaneceu sob inundação por 60,73 segundos em média, valor que corresponde à duração integral do disparo e confirma que, sem mitigação, nada interrompe a inundação antes do encerramento do ataque pelo agressor. As três estratégias de ancoragem reduziram a exposição para a faixa de poucos segundos, com eficácia superior a 89% em todos os casos.

Tabela 4.9: Síntese da eficácia das estratégias de ancoragem sob inundação GRE distribuída de origem real. Intervalos de confiança de 95% pela distribuição t de Student com vinte e nove graus de liberdade ($n = 30$ por condição).

Estratégia	**Âncora**	**Exposição (s)**	**IC 95%**	**Redução**
off (referência)	—	60,73	[60,57; 60,90]	—
ip	`IPV4_SRC`	3,38	[2,62; 4,14]	94,4%
mac	`ETH_SRC`	3,91	[3,01; 4,80]	93,6%
auto	`IPV4_SRC`	6,28	[4,22; 8,33]	89,7%

A estratégia *ip* reduziu a exposição para 3,38 segundos em média, com intervalo de confiança entre 2,62 e 4,14 segundos, o que representa uma redução de 94,4% em relação à referência. O resultado contrasta de forma direta com o observado no vetor ACK, em que a mesma estratégia não produziu qualquer mitigação. A diferença tem causa única e já antecipada. Como a origem do GRE é real e estável, a regra ancorada em `IPV4_SRC` casa os pacotes do agressor e os descarta no plano de dados. O registro forense confirma o mecanismo, com noventa bloqueios diretos por `IPV4_SRC` ao longo das trinta repetições, trinta para cada origem, e nenhuma ocorrência de bloqueio por endereço físico. Esse desfecho reproduz, em um protocolo distinto, o que já se observara no vetor UDP, e sustenta a tese de que a eficácia do bloqueio por endereço de origem depende da estabilidade da origem, e não da família do protocolo.

A estratégia *mac* reduziu a exposição para 3,91 segundos em média, com intervalo de confiança entre 3,01 e 4,80 segundos, uma redução de 93,6%. A âncora em `ETH_SRC` é igualmente eficaz, pois o endereço físico de origem é estável por construção, e os noventa bloqueios registrados ocorreram diretamente por endereço físico, sem escalonamento. A proximidade entre as exposições das estratégias *ip* e *mac* neste vetor é coerente com a natureza da origem. Quando o endereço IP é confiável, ambas as âncoras casam o tráfego no primeiro ciclo de atuação, e a pequena diferença entre as médias situa-se dentro da variação esperada do ciclo de varredura.

A estratégia *auto* reduziu a exposição para 6,28 segundos em média, com intervalo de confiança entre 4,22 e 8,33 segundos, uma redução de 89,7%. O registro forense mostra noventa e três bloqueios, todos ancorados em `IPV4_SRC`, e nenhuma escalada para o endereço físico ao longo das trinta repetições. O total de noventa e três decompõe-se em noventa bloqueios iniciais, três por origem, somados a re-bloqueios pontuais nas repetições de contenção incremental, em que uma origem normalizada voltou a cruzar o limiar antes do término do disparo. Esse comportamento confirma a coerência da estratégia adaptativa diante de origem confiável. Ao constatar que o contador da regra por `IPV4_SRC` cresce, sinal de que o descarte é efetivo, o orquestrador não infere volatilidade de origem e permanece na âncora de camada de rede, sem o passo de escalonamento que caracterizou o vetor ACK. O mesmo padrão fora observado no vetor UDP de origem fixa, e a sua repetição sob o protocolo GRE reforça que a escalada para o endereço físico é reservada, por construção, ao caso em que a regra por IP se mostra inerte.

A exposição média da estratégia *auto* é superior à das duas estratégias diretas, e o seu intervalo de confiança é visivelmente mais amplo. Essa dispersão não decorre de ineficácia, e sim do caráter distribuído do ataque combinado ao modo de operação por origem do mecanismo de descarte. Como os três dispositivos atacam em paralelo, e a detecção de cada origem depende do instante em que sua taxa cruza o limiar de inundação em relação ao ciclo de varredura, o bloqueio das três origens nem sempre se completa no mesmo ciclo. Em parte das repetições, uma das origens foi contida ciclos depois das demais, e o tráfego residual dessa origem, ainda não bloqueada, manteve o alvo em inundação por alguns segundos adicionais. O tempo de exposição total é, portanto, governado pela última origem a ser bloqueada, e não pela primeira. Esse efeito é próprio de cenários distribuídos e constitui uma observação relevante, pois evidencia que, sob ataque de múltiplas origens, a contenção se completa de forma incremental, à medida que cada origem é identificada e descartada de modo independente.

A Figura 4.3 contrasta a persistência do ataque na condição de referência com a contenção obtida pela estratégia *auto* em uma repetição cuja exposição coincide com a mediana das trinta. As séries são extraídas diretamente da telemetria da vítima, amostra a amostra, sem suavização, o que preserva a variação natural da taxa de pacotes característica de uma inundação real. Sem mitigação, a vítima permanece sob inundação por toda a janela do disparo. Sob a estratégia *auto*, o bloqueio por `IPV4_SRC` casa o tráfego de origem confiável e contém o ataque em poucos segundos, sem necessidade de escalada para o endereço físico.

O caráter incremental da contenção em ataque distribuído fica visível na fração de repetições da estratégia *auto* em que uma das origens foi bloqueada com atraso em relação às demais. A Figura 4.4 apresenta uma dessas repetições, na qual a taxa no alvo cai do pico do ataque para um patamar residual sustentado por uma única origem ainda ativa, antes de retornar a zero quando essa última origem é finalmente descartada. O patamar intermediário, próximo a um terço da intensidade do ataque distribuído, é coerente com a contribuição de uma das três origens, e não representa falha da mitigação, e sim a sua conclusão escalonada por dispositivo.

A inundação GRE distribuída ofereceu ainda a oportunidade de exercitar a propagação de regras

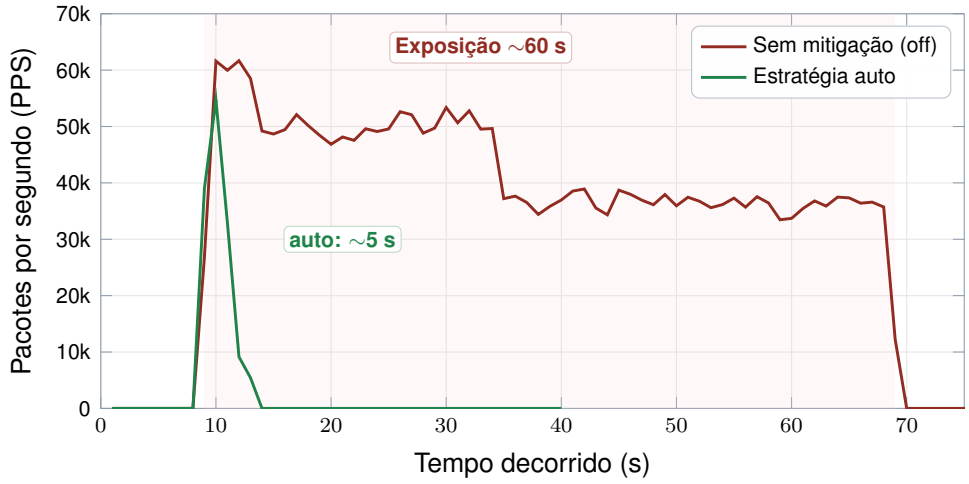

Figura 4.3: Taxa de pacotes por segundo na interface da vítima durante a inundação GRE distribuída de origem real.

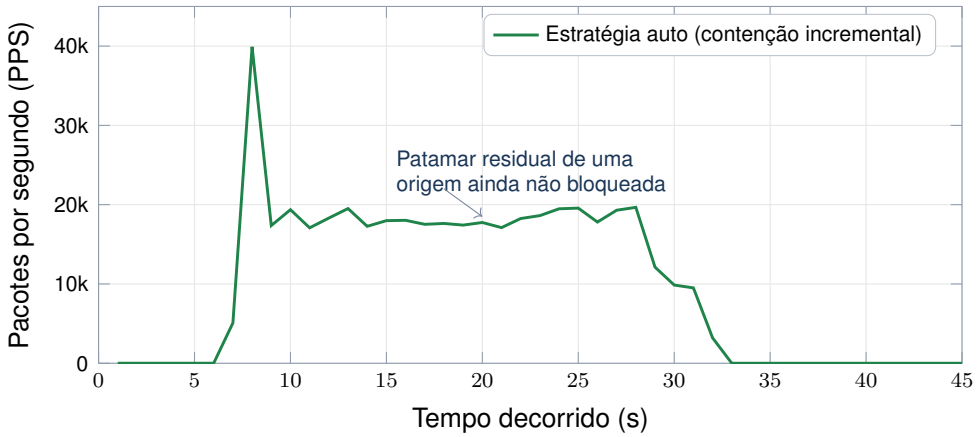

Figura 4.4: Repetição da estratégia *auto* sob inundação GRE distribuída com bloqueio tardio de uma das origens.

na topologia de dois comutadores descrita na Seção 3.1.4. Em cada bloqueio, o registro forense do orquestrador documenta a sincronização da regra de descarte nos dois comutadores do domínio de encaminhamento, identificados por `of:00003c2c30250880` e `of:00003c2c3024ef80`, o que confirma que a ação de mitigação foi instalada de forma consistente em ambos os comutadores. Esse registro fornece evidência empírica, sob carga real de ataque, da extensão da arquitetura a dois comutadores, antes apresentada apenas em termos de projeto.

4.6.4 Posição do Vetor GRE no Quadro Experimental

O vetor GRE consolida a leitura construída ao longo do capítulo por um caminho independente. Ao reproduzir, em um protocolo de encapsulamento distinto e sob regime distribuído, a mesma eficácia do bloqueio por endereço de origem já observada nos vetores SYN e UDP, o ensaio reforça que a estratégia *ip* é plenamente adequada quando a origem é confiável, qualquer que seja a família do protocolo. Ao exibir, na estratégia *auto*, a permanência na âncora de camada de rede sem escalonamento, repete o comportamento do vetor UDP e demonstra que a seleção da âncora pela estratégia adaptativa responde à volatilidade observável da origem, e não a uma preferência fixa por um critério de correspondência. Por fim, ao expor a contenção incremental sob ataque de múltiplas origens, o vetor acrescenta uma observação própria sobre o comportamento da defesa em cenários distribuídos, que não havia sido evidenciada pelos demais vetores. Em conjunto com o vetor ACK, em que a origem forjada exige a escalada para o endereço físico, o GRE fecha o contraste que sustenta a contribuição central deste trabalho, a saber, que a âncora de mitigação eficaz é determinada pela propriedade de origem do ataque, observável em tempo de execução, e não pelo protocolo empregado.

4.7 MITIGAÇÃO SOB INUNDAÇÃO UDP DE ORIGEM FIXA

Os vetores anteriores cobriram inundações TCP, de origem fixa no SYN e forjada no ACK, e a inundação GRE de origem real sob regime distribuído. Esta seção avalia o vetor UDP, que completa o quadro experimental por acrescentar uma família de protocolo distinta de transporte e um cenário de origem estável submetido à defesa. O ataque UDP foi disparado pelos três *bots* a partir de seus endereços reais, sem falsificação. O desenho experimental reproduz a estrutura de quatro condições dos demais vetores, com a condição de referência sem mitigação (*off*) e as três estratégias de ancoragem (*ip*, *mac* e *auto*) descritas na Subseção 4.5.2. A repetição das três estratégias, mesmo em um vetor de origem confiável em que a estratégia *ip* já é suficiente, verifica se a estratégia adaptativa permanece na âncora de camada de rede sem escalar para o endereço físico, em coerência com o observado nos vetores SYN e GRE. Cada condição foi repetida trinta vezes, com período de varredura de um segundo.

A Tabela 4.10 consolida a eficácia das quatro condições, com as médias de tempo de exposição do alvo e os respectivos intervalos de confiança de 95%, estimados pela distribuição t de Student.

O tamanho amostral de cada condição consta na tabela, após a exclusão das capturas inválidas descritas no Apêndice D.

Tabela 4.10: Síntese da eficácia das estratégias de ancoragem sob UDP *flood* de origem fixa. Intervalos de confiança de 95% pela distribuição t de Student. O tamanho amostral n por condição consta na coluna respectiva.

Estratégia	**Âncora**	n	**Exposição (s)**	**IC 95%**	**Redução**
off (referência)	—	29	25,55	[24,96; 26,14]	—
ip	`IPV4_SRC`	30	2,27	[2,05; 2,48]	91,1%
mac	`ETH_SRC`	30	2,13	[1,92; 2,35]	91,7%
auto	`IPV4_SRC`	30	1,90	[1,72; 2,08]	92,6%

Na condição de referência, o alvo permaneceu sob inundação por 25,55 segundos em média, com intervalo de confiança de 95% entre 24,96 e 26,14 segundos. Esse valor aproxima-se do observado no vetor SYN de origem fixa e contrasta com os cerca de sessenta segundos do ACK forjado. A diferença tem a mesma raiz já discutida. Como a origem do UDP é estável, o *firewall* de borda contém parte do tráfego pela filtragem de estado, e o registro do OPNsense mostra a alternância de eventos de bloqueio e liberação por endereço de origem que caracteriza essa contenção parcial. O que escapa do filtro de perímetro mantém o alvo em *flood* pela janela observada.

As três estratégias de mitigação reduziram a exposição para a faixa de dois segundos, com eficácia entre 91,1% e 92,6%. A estratégia *ip* reduziu a exposição para 2,27 segundos em média, com intervalo de confiança entre 2,05 e 2,48 segundos, uma redução de 91,1%. A regra ancorada em `IPV4_SRC` casou o tráfego, pois a origem é estável, e os contadores de descarte cresceram desde o primeiro ciclo. As estratégias *mac* e *auto* alcançaram exposições de 2,13 e 1,90 segundos, com reduções de 91,7% e 92,6%. O registro forense da estratégia *auto* confirma que o modo adaptativo jamais precisou migrar para o endereço físico, em coerência com o vetor de origem confiável. O registro também exibe o tratamento correto do gateway de borda, identificado como pertencente à *allowlist* e preservado a cada ciclo, o que evita que a infraestrutura de roteamento seja bloqueada por engano. A Figura 4.5 compara a condição de referência com a contenção obtida pela estratégia *auto*. As curvas reproduzem a telemetria do alvo amostra a amostra, sem suavização, na repetição de exposição mediana. Como a origem é estável, o bloqueio por `IPV4_SRC` casa o tráfego e contém o ataque em poucos segundos, sem necessidade de escalada para o endereço físico. O eixo do tempo é apresentado até 75 segundos; a série completa registra o retorno do tráfego ao patamar de fundo após o `UNBLOCK`.

Esse resultado tem uma leitura que transcende o vetor UDP em si. Ele demonstra que o bloqueio por endereço de origem não é uma técnica deficiente. Contra origem fixa, o bloqueio por IP é plenamente eficaz, com redução de 91,1%, valor próximo dos 91,8% obtidos no vetor SYN, também de origem estável. A falha do bloqueio por IP, registrada na avaliação do ACK, manifesta-se apenas sob falsificação de origem, e não como uma deficiência intrínseca do critério. Lido em conjunto com a Seção 4.5, o vetor UDP esclarece a natureza da contribuição proposta. O mérito da estratégia *auto* não está em substituir o endereço IP pelo endereço físico, e sim em

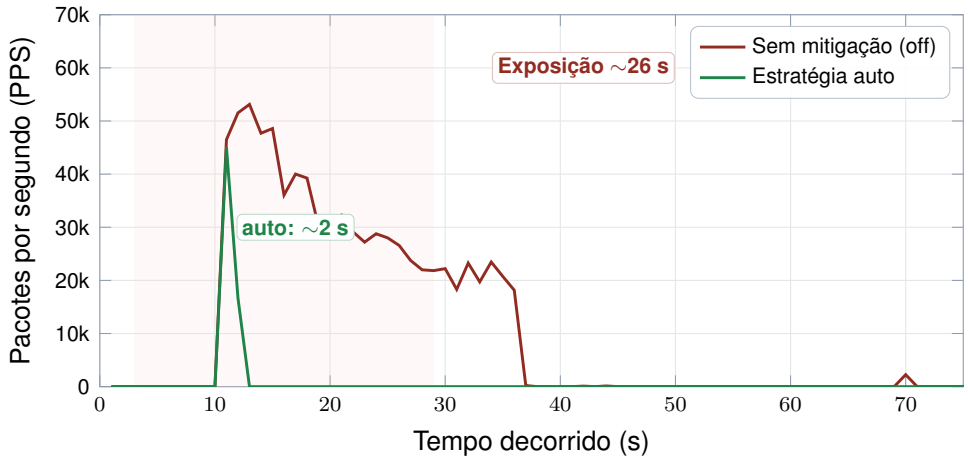

Figura 4.5: Taxa de pacotes por segundo na interface da vítima durante o UDP *flood* de origem fixa.

selecionar a âncora adequada a partir da evidência observável em tempo de execução. O endereço IP permanece como caminho primário porque é o critério correto quando a origem é confiável, e a escalada para o endereço físico é reservada para o caso em que a origem se revela volátil. O UDP de origem fixa é a evidência de que essa escolha primária se justifica.

4.8 SÍNTESE DOS RESULTADOS

Os resultados reunidos neste capítulo sustentam conclusões complementares sobre quatro vetores volumétricos da família Mirai, avaliados sob diferentes propriedades de origem e protocolo.

A primeira diz respeito à eficácia da arquitetura contra ataques de origem estável. No SYN *flood*, a mitigação reduziu o tempo de exposição da vítima de 24,50 para 2,00 segundos, redução de 91,8%. No UDP *flood*, também de origem fixa, a redução foi de 25,55 para 2,27 segundos, 91,1%. No GRE *flood* distribuído de origem real, o bloqueio ancorado no endereço IP reduziu a exposição de 60,73 para 3,38 segundos, redução de 94,4%. Em todos esses casos o bloqueio ancorado no endereço IP casa o tráfego do agressor, e a latência é governada pelo ciclo de varredura. O bloqueio por endereço de origem, quando a origem é confiável, é uma defesa eficaz e suficiente, qualquer que seja a família do protocolo.

A segunda conclusão, de natureza metodológica, é o alerta de que a eficiência algorítmica de um controlador SDN tem pouco valor prático quando o hardware adjacente não consegue sincronizar os comandos durante um ataque volumétrico. O fenômeno das regras órfãs no plano de dados, não observado nas validações em simuladores, foi caracterizado e tratado pelo verificador de consistência em malha fechada, que restabeleceu a sincronização entre o estado lógico do controlador e o estado físico do comutador. A coleta do vetor GRE, conduzida sobre a topologia de dois comutadores, registrou ainda a propagação consistente das regras de descarte em ambos os comutadores do domínio de encaminhamento, o que estende a verificação de consistência ao cenário de dois comutadores.

A terceira conclusão decorre da avaliação sob falsificação de origem. Quando o atacante torna o endereço IP de origem volátil, o bloqueio convencional por IP perde a eficácia, sem reduzir a exposição de forma estatisticamente perceptível. A identidade estável do agressor, contudo, permanece observável na camada de enlace, pois o controlador agrega o tráfego pelo endereço físico, sob o qual os endereços forjados ficam ocultos. O orquestrador explora essa âncora ao escalar o bloqueio para o endereço físico em 2,13 segundos quando constata a ineficácia da regra por IP, reduzindo a exposição em 92,7% na estratégia adaptativa e em 96,7% no bloqueio direto por endereço físico. A escolha entre as âncoras não é fixa, e sim guiada pela evidência observável em tempo de execução, o que preserva o bloqueio por IP como caminho primário e reserva a escalada para o caso forjado. Os vetores de origem real, em especial o GRE avaliado nas três estratégias, confirmam essa leitura por um caminho independente: diante de origem confiável, a estratégia adaptativa permanece na âncora de rede sem escalonamento, o que demonstra que a seleção da âncora responde à propriedade de origem do ataque, e não ao protocolo empregado.

4.8.1 Análise Estatística Complementar

Os tempos de exposição possuem limite superior associado à duração do disparo, o que produz distribuições assimétricas nas condições próximas a esse limite. Por essa razão, além da média e do intervalo de confiança apresentados nas seções anteriores, reporta-se um conjunto de medidas robustas à assimetria e um teste de hipótese não paramétrico. A Tabela 4.11 consolida, por vetor e estratégia, a mediana, o intervalo interquartílico, a amplitude e o intervalo de confiança de 95% obtido por reamostragem *bootstrap*.

A comparação entre a condição de referência e cada estratégia de ancoragem apoiou-se no teste de Mann-Whitney U (bicaudal, $\alpha = 0{,}05$), escolhido pela assimetria das distribuições, e no delta de Cliff (δ) como medida de tamanho de efeito. Em quinze das dezesseis comparações a redução do tempo de exposição foi estatisticamente significativa, com separação completa entre as amostras ($p < 0{,}001$; $\delta = +1{,}00$), o que sustenta a eficácia das estratégias nos vetores SYN, UDP e GRE, bem como das âncoras física e adaptativa no vetor ACK. A exceção é o bloqueio por endereço IP sob a inundação ACK com origem forjada, em que a diferença em relação à referência não foi estatisticamente significativa ($U = 568$; $p = 0{,}079$) e o tamanho de efeito foi pequeno ($\delta = +0{,}26$), com redução relativa de apenas 1,0%. Esse resultado deve ser interpretado como ausência de eficácia prática da ancoragem por IP quando a origem é forjada, e não como uma estratégia apenas menos eficiente, pois o identificador de rede deixa de individualizar a origem do ataque.

O conjunto formado pela mitigação rápida contra origem estável, pela consistência verificada do plano de dados, inclusive no cenário de dois comutadores, e pela ancoragem adaptativa resistente à falsificação de origem constitui o núcleo da contribuição técnica deste trabalho.

Tabela 4.11: Estatística descritiva complementar do tempo de exposição do alvo (em segundos), por vetor e estratégia de ancoragem. As medidas de mediana, intervalo interquartílico e amplitude são robustas à assimetria das distribuições, limitadas superiormente pela duração do disparo. O intervalo de confiança de 95% é obtido por reamostragem *bootstrap* (10.000 reamostras, método do percentil).

Vetor	**Estratégia**	n	**Mediana**	**Q1–Q3**	**Mín–Máx**	**IC 95% (*bootstrap*)**
SYN	off	26	23	22–24	21–48	[22,81; 26,31]
SYN	ip	30	2	2–2	1–3	[1,83; 2,17]
SYN	mac	30	2	2–3	1–5	[2,13; 2,77]
SYN	auto	29	2	2–3	1–14	[2,07; 3,55]
UDP	off	29	26	25–26	22–29	[25,00; 26,10]
UDP	ip	30	2	2–3	1–3	[2,07; 2,47]
UDP	mac	30	2	2–2	1–3	[1,93; 2,33]
UDP	auto	30	2	2–2	1–3	[1,73; 2,07]
GRE	off	30	61	60–61	60–61	[60,57; 60,87]
GRE	ip	30	3	3–3	2–13	[2,80; 4,17]
GRE	mac	30	3	2–4	2–11	[3,13; 4,80]
GRE	auto	30	5	3–7	2–26	[4,53; 8,33]
ACK	off	30	58	56–60	32–60	[54,63; 58,33]
ACK	ip	30	57	55–58	47–60	[54,93; 57,30]
ACK	mac	30	2	2–2	1–3	[1,70; 2,00]
ACK	auto	30	4	4–4	3–5	[3,97; 4,37]

5. CONCLUSÃO

Esta dissertação investigou a viabilidade operacional de uma defesa de borda orientada por Redes Definidas por Software para a mitigação de ataques volumétricos da família Mirai, sob a restrição de que a validação ocorresse em *hardware* comercial físico, e não em ambiente emulado. O percurso confirmou que a programabilidade do paradigma SDN, embora necessária, não basta. A eficácia prática depende da capacidade do plano de dados físico de sincronizar, em tempo de ataque, as decisões emitidas pelo controlador. O ambiente *Hardware-in-the-Loop* construído em torno do controlador ONOS e do comutador Dell S3048-ON, operando sob SmartFabric OS10 e OpenFlow 1.3, revelou a persistência de regras órfãs no plano de dados do comutador, comportamento não observado nas validações conduzidas em simuladores e capaz de comprometer de forma silenciosa a disponibilidade da rede em produção. A pesquisa caracterizou esse comportamento e propôs, implementou e avaliou um mecanismo de compensação em software que reconciliou os eventos de inconsistência identificados entre os planos de controle e de dados. A avaliação estendeu-se a quatro vetores volumétricos da família Mirai: as inundações SYN e UDP de origem fixa, a inundação ACK com falsificação dinâmica de origem e a inundação GRE de origem real distribuída, esta conduzida sobre a topologia estendida de dois comutadores. Esse conjunto permitiu testar a defesa tanto na sua trajetória esperada quanto nas condições adversas em que o identificador de rede deixa de ser confiável e em que o caminho de encaminhamento se distribui por mais de um equipamento.

5.1 SÍNTESE DO CUMPRIMENTO DOS OBJETIVOS

A pesquisa cumpriu os objetivos específicos estabelecidos no Capítulo 1. Para cada um, registra-se a seguir a evidência correspondente.

O levantamento do estado da arte resultou na revisão apresentada no Capítulo 2, que comparou trabalhos representativos segundo critérios de ambiente de validação, uso de *hardware* real e investigação de regras órfãs. Essa análise delimitou a lacuna central do trabalho, a saber, a escassez de validações em equipamentos físicos comerciais e a ausência de tratamento do fenômeno de persistência de regras no plano de dados.

A estruturação da topologia de experimentação concretizou-se no ambiente *Hardware-in-the-Loop* descrito no Capítulo 3, que integrou o controlador ONOS 2.7 LTS ao comutador Dell S3048-ON por OpenFlow 1.3, com segmentação fora de banda entre os planos de controle, de dados e de ataque, e foi posteriormente estendido a um segundo comutador do mesmo modelo, interligado por enlace dedicado sob um único domínio de encaminhamento. A operação efetiva das sessões OpenFlow foi verificada no registro de dispositivos do controlador.

A reprodução de cenários de ameaça materializou quatro vetores volumétricos da família Mirai, despachados por dispositivos infectados sob orquestração de um servidor de comando e controle. O vetor SYN submeteu o alvo a picos da ordem de cinquenta mil pacotes por segundo, com registros individuais superiores a setenta mil, a partir de origem fixa. O vetor UDP repetiu a condição de origem fixa em outra família de protocolo. O vetor ACK acionou a falsificação dinâmica de origem, em que cada pacote trafega com um endereço de rede distinto. O vetor GRE, por fim, levou a avaliação à topologia de dois comutadores. O ataque partiu dos três *bots* locais sobre o SW-A, com origem real distribuída entre endereços não forjados, e a regra de bloqueio precisou efetivar-se de forma sincronizada nos dois comutadores. O *bot* remoto hospedado no segundo comutador não participou da inundação GRE. Esse nó serviu à demonstração da janela de indisponibilidade reativa entre comutadores distintos (Seção 3.1.4.1).

A caracterização do comportamento do *hardware* identificou, sob estresse volumétrico, a assincronia entre a remoção lógica de regras no controlador e a sua efetiva exclusão no plano de dados do equipamento, que origina o fenômeno das regras órfãs na ausência de compensação. A extensão multi-switch acrescentou a essa caracterização a janela transitória de indisponibilidade do encaminhamento reativo, observada na primeira comunicação entre hospedeiros de comutadores distintos (Seção 3.1.4.1).

O desenvolvimento de mecanismos de compensação materializou-se no módulo DDSO e em seu verificador de consistência em malha fechada, que substitui a confiança no sucesso lógico reportado pelo controlador pela auditoria ativa dos contadores de *hardware*. A esse mecanismo somou-se a lógica de ancoragem adaptativa da regra de descarte, capaz de migrar do endereço IP para o endereço físico quando a evidência indica origem forjada.

A validação de desempenho comprovou a eficácia da solução nos vetores avaliados. Na inundação SYN, o tempo de exposição do alvo caiu de 24,50 segundos, na referência, para 2,00 segundos sob mitigação, redução de 91,8%, com intervalos de confiança de 95% sem sobreposição. Na inundação UDP de origem fixa, a exposição caiu de 25,55 para 2,27 segundos, redução de 91,1%, valor que reforça a eficácia do bloqueio por endereço quando a origem é confiável. Na inundação ACK com origem forjada, cuja exposição de referência alcançou 56,77 segundos, na prática a duração integral do disparo, o bloqueio convencional por endereço IP mostrou-se inócuo: a exposição de 56,20 segundos, com intervalo de confiança sobreposto ao da referência, não apresentou redução estatisticamente significativa (Tabela 4.6). A ancoragem pelo endereço físico, em contraste, reduziu a exposição para 1,87 segundos e a estratégia adaptativa para 4,17 segundos, reduções de 96,7% e 92,7%. A escalada da regra de IP para a regra de enlace ocorreu em 2,13 segundos em média, com intervalo de confiança de 95% de [2,13; 2,14] segundos, comportamento coerente com o parâmetro de projeto que define um ciclo de evidência antes da escalada. Na inundação GRE, conduzida a partir das origens reais distribuídas entre os *bots* locais sobre o SW-A, com a regra de bloqueio sincronizada na topologia de dois comutadores e trinta repetições por condição, a exposição de referência de 60,73 segundos caiu para 3,38 segundos com o bloqueio por endereço IP (redução de 94,4%), para 3,91 segundos com a âncora física (93,6%) e para 6,28 segundos com a estratégia adaptativa (89,7%). Sem evidência de origem forjada, a

estratégia adaptativa não escalou, mantendo o bloqueio por endereço IP, e a regra de bloqueio foi registrada nos dois comutadores. Nesse cenário de múltiplas origens reais, a exposição mostrou-se governada pela última origem bloqueada, caracterizando a contenção incremental por origem. Com o verificador de consistência ativo, a sincronização entre o estado lógico do controlador e o estado físico do comutador foi restabelecida ao final de cada evento de mitigação avaliado, reconciliando as regras órfãs identificadas nesses eventos.

Com isso, considera-se cumprido também o objetivo geral: a arquitetura proposta foi implementada e avaliada empiricamente em ambiente *Hardware-in-the-Loop*, mitigou de forma automatizada os vetores volumétricos avaliados e reconciliou, por compensação em software, as falhas de sincronização e de persistência de regras identificadas no plano de dados físico nos eventos de mitigação avaliados.

5.2 RESPOSTA ÀS QUESTÕES DE PESQUISA

Os resultados consolidados permitem responder às três questões formuladas na Seção 1.1 e confrontar as hipóteses correspondentes com a evidência experimental.

Quanto à **QP1**, a validação em *hardware* comercial revelou falhas que as validações em emuladores não alcançam. A persistência de regras órfãs no plano de dados, observada nas versões iniciais do orquestrador (Seção 4.4), associa-se à defasagem entre a confirmação lógica da remoção e a sua efetivação no equipamento, divergência não capturada por uma tabela de fluxos mantida em memória por um emulador. A janela de indisponibilidade do encaminhamento reativo no caminho entre dois comutadores (Seção 3.1.4.1) integra o mesmo quadro. A hipótese **H1** foi corroborada quanto à ocorrência de regras órfãs no *hardware* comercial avaliado; a comparação sistemática com ambientes emulados, contudo, não integrou o desenho experimental, de modo que a sua não manifestação em emulação permanece indicação apoiada na literatura, e não verificação conduzida neste trabalho.

Quanto à **QP2**, o verificador de consistência em malha fechada demonstrou que a reconciliação entre os planos é alcançável exclusivamente em software, sem modificação do equipamento. Ao auditar os contadores de fluxo do comutador e reemitir comandos de remoção até a confirmação no plano de dados, o mecanismo restabeleceu a sincronização ao final de cada evento de mitigação avaliado, inclusive nos eventos do vetor GRE com regras instaladas nos dois comutadores, com latência de mitigação de 0,71 segundo dominada pelo ciclo de varredura de um segundo (Tabela 4.4). A hipótese **H2** foi corroborada, com a ressalva de que a taxa de ocorrência de regras órfãs e o tempo de reconciliação não foram objeto de medição sistemática, conforme registrado na Seção 5.4.

Quanto à **QP3**, a ancoragem adaptativa preservou a eficácia da mitigação na condição em que o endereço IP deixou de ser confiável: a regra ancorada no endereço físico conteve o vetor que o bloqueio convencional não conteve, com reduções de 96,7% e 92,7% nas estratégias direta e adaptativa (Tabela 4.6). A hipótese **H3** foi corroborada, com uma delimitação de escopo

demonstrada empiricamente: a âncora física é posicionalmente dependente, válida no segmento de ingresso, a montante do salto de camada 3, e contraindicada a jusante, onde o endereço observado passa a ser o do roteador (Seção 3.1.5).

5.3 PRINCIPAIS CONTRIBUIÇÕES E INOVAÇÕES

As contribuições desta dissertação organizam-se em três eixos.

Contribuição metodológica. A adoção da abordagem *Hardware-in-the-Loop* sobre comutadores comerciais baseados em ASIC permitiu observar comportamentos do plano de dados físico que não são capturados pelas validações em emuladores como o Mininet. O fenômeno das regras órfãs, documentado e caracterizado neste trabalho, constitui evidência de que a correção lógica de um controlador SDN não garante, por si só, a segurança da infraestrutura quando o *hardware* subjacente não consegue acompanhar, em tempo real, as instruções de remoção de regras. A extensão do ambiente a dois comutadores agregou duas evidências do mesmo gênero: a janela de indisponibilidade do encaminhamento reativo, que tende a crescer com o número de saltos, e a reescrita do endereço físico de origem no salto de camada 3, verificada por captura simultânea nos dois segmentos do caminho. Esse conjunto constitui um alerta concreto sobre os limites da validação puramente simulada.

Contribuição tecnológica. O orquestrador DDSO, desenvolvido como aplicação externa em Python e dotado do verificador de consistência em malha fechada, demonstrou ser uma estratégia viável e resiliente para a defesa de borda. Ao auditar de forma ativa o plano de dados e reemitir comandos de remoção até a confirmação da sincronização, o verificador atua como camada de reconciliação que protege o operador contra falhas silenciosas de persistência durante incidentes. A separação entre o orquestrador e o núcleo do controlador, além de isolar falhas, não comprometeu o regime de latência de mitigação reportado no Capítulo 4.

Contribuição operacional. A avaliação sob falsificação dinâmica de origem revelou que o identificador de camada de rede, base das defesas convencionais por bloqueio de IP, perde valor quando o atacante o torna volátil. O trabalho mostrou que a aplicação de encaminhamento reativo do controlador agrega o tráfego pelo endereço físico, e não pelo endereço IP, de modo que os endereços forjados sequer se tornam visíveis ao plano de controle. Disso resulta uma âncora de identidade estável e observável na borda, o endereço físico de origem. O orquestrador foi dotado de lógica capaz de inferir a falsificação a partir do comportamento dos contadores de *hardware* e de escalar o bloqueio do endereço IP para o endereço físico em tempo de execução, sem depender de conhecimento prévio sobre a natureza do ataque. A validação experimental confirmou que essa escalada contém o vetor forjado que o bloqueio convencional não conteve, e que ela preserva o bloqueio por IP como caminho primário, eficaz e preferível quando a origem é confiável. Confirmou, ainda, que a sua aplicação exige consciência de topologia, dado que a âncora física só identifica o dispositivo no segmento anterior ao salto de camada 3.

5.4 LIMITAÇÕES DA PESQUISA

Os resultados devem ser interpretados à luz das limitações do desenho experimental.

As campanhas estatísticas dos vetores SYN, ACK e UDP empregaram um único comutador físico (SW-A), e a campanha do vetor GRE estendeu a coleta à topologia de dois comutadores do mesmo modelo e fabricante. A diversidade de ASICs entre plataformas e a variação de capacidade de TCAM permanecem, contudo, fora do escopo, de modo que o comportamento de persistência aqui observado é específico da família de equipamento avaliada.

A geração de tráfego dos nós atacantes foi virtualizada. O *hardware* no laço de validação foi o comutador comercial físico, elemento sob avaliação, enquanto os nós atacantes, o alvo e o controlador foram implementados como máquinas virtuais. Dispositivos físicos de borda do tipo Raspberry Pi, embora presentes na infraestrutura, não foram empregados como fontes de tráfego nos ensaios aqui reportados.

A ancoragem pelo endereço físico, eficaz contra a falsificação de origem, traz duas restrições de escopo que delimitam o seu alcance. A primeira é a dependência do domínio de camada de enlace, demonstrada empiricamente na Seção 3.1.5: a jusante do salto de camada 3, o endereço físico observado é o do roteador, e uma regra ali ancorada bloquearia indiscriminadamente o tráfego legítimo que transita pelo *gateway*, razão pela qual o orquestrador restringe essa âncora ao segmento de ingresso. A segunda é a granularidade. A regra por endereço físico descarta todo o tráfego do hospedeiro de origem, e não apenas o tráfego de ataque. Para um dispositivo dedicado ao ataque, isso é aceitável, mas em um hospedeiro legítimo comprometido equivaleria a uma negação de serviço sobre os seus serviços benignos. Soma-se a essas restrições o fato de a falsificação do próprio endereço físico pelo atacante não ter sido avaliada: sob *spoofing* de MAC, a âncora física perderia a estabilidade que a fundamenta, hipótese que remete às camadas preventivas de validação de origem discutidas nos trabalhos futuros.

O controlador operou em instância única. Embora o ONOS ofereça arquitetura distribuída com alta disponibilidade, os aspectos de *cluster*, replicação de estado, eleição de líder e recuperação após a falha de um nó não integraram esta avaliação, permanecendo como investigação futura (Seção 5.5).

A detecção apoiou-se em limiares fixos. Embora eficaz contra a volumetria bruta dos vetores avaliados, essa estratégia pode exigir refinamento diante de ataques de baixa taxa, que mimetizam o comportamento de usuários legítimos.

A análise estatística apoiou-se em trinta repetições por condição nos quatro vetores, sumarizadas por média e intervalo de confiança de 95%. Os tempos de exposição, contudo, possuem limite superior associado à duração do disparo, o que produz distribuições assimétricas nas condições próximas a esse limite; nesses casos, a não sobreposição de intervalos de confiança não substitui, por si só, um teste de hipótese específico, cuja incorporação, ao lado de estatísticas não paramétricas como a mediana e os quartis, reforçaria a caracterização das diferenças observadas. Permanece em aberto, ainda, a quantificação da persistência: o fenômeno das regras órfãs foi identificado

e registrado de forma qualitativa na fase inicial do desenvolvimento, e a correção incorporada ao orquestrador o reconciliou nos eventos subsequentes avaliados, mas a taxa de ocorrência e o tempo de reconciliação do plano de dados não foram objeto de medição sistemática, o que constitui consolidação a cargo de trabalho futuro.

Por fim, os resultados referem-se a uma combinação específica de sistema operacional, *firmware* e controlador. O comportamento de sincronização e de persistência observado pode variar em outras versões ou em outros modelos de *hardware*, o que recomenda cautela na generalização.

5.5 TRABALHOS FUTUROS

A partir dos resultados e das limitações identificadas, vislumbram-se os seguintes desdobramentos.

Consolidação experimental da persistência de regras. Conduzir campanha dedicada à medição da taxa de ocorrência de regras órfãs e do tempo de reconciliação do plano de dados, comparando de forma sistemática as versões do orquestrador com e sem o verificador de consistência ativo, com número ampliado de repetições e com a substituição dos geradores de tráfego virtualizados pelos nós físicos Raspberry Pi já integrados à infraestrutura.

Ancoragem além da borda de enlace. A dependência posicional da âncora física, demonstrada na Seção 3.1.5, delimita a atuação do orquestrador ao segmento de ingresso. Cabe investigar estratégias complementares para o tráfego forjado que cruza o salto de camada 3, em particular a validação de endereço de origem na borda (SAV, RFC 7039; BCP 38), como camada preventiva que reduziria, a montante, o próprio volume de tráfego com origem falsificada.

Mitigação por fluxo na origem comprometida. A ancoragem pelo endereço físico resolve a falsificação ao custo de bloquear todo o tráfego do hospedeiro. Investigar critérios de granularidade mais fina, que distingam o tráfego de ataque do tráfego benigno de um mesmo dispositivo, ampliaria a aplicabilidade da defesa a hospedeiros legítimos comprometidos.

Plano de dados programável e diversidade de *hardware*. Investigar se a migração do OpenFlow para a linguagem P4 reduziria a ocorrência de regras órfãs, ao permitir que a própria lógica de auditoria de estado seja processada no chip do comutador, e repetir o protocolo *Hardware-in-the-Loop* com equipamentos de outros modelos e fabricantes, em especial com maior capacidade de TCAM, a fim de quantificar o impacto da memória sobre a persistência. Nesse mesmo eixo, a avaliação de um conjunto distribuído de controladores ONOS permitiria mensurar como a latência de sincronização entre nós afeta a consistência do plano de dados em larga escala.

Detecção adaptativa e inteligência de ameaças. Evoluir o motor de detecção do DDSO para modelos de aprendizado de máquina de baixo custo computacional executados no próprio orquestrador, dispensando a calibração manual de limiares e ampliando a cobertura para vetores de baixa taxa, e integrar o orquestrador a fontes de inteligência de ameaças para o pré-carregamento de

regras contra servidores de comando e controle conhecidos, reduzindo o intervalo de exposição durante a rampa inicial do ataque.

O título desta dissertação anuncia desafios de sincronização e de persistência de regras. Ao término do percurso, é possível nomeá-los com precisão: a janela transitória de indisponibilidade do encaminhamento reativo, que tende a se agravar com o número de saltos; a persistência de regras órfãs no plano de dados após a remoção lógica; e a dependência posicional da âncora física de bloqueio, válida a montante do salto de camada 3 e contraindicada a jusante. A programabilidade das Redes Definidas por Software amplia o repertório de defesa contra ameaças volumétricas, mas a sua eficácia em produção depende da integridade da execução no plano de dados físico e da escolha de um identificador de bloqueio que resista às táticas de evasão do atacante. Garantir que as decisões do controlador sejam efetivamente aplicadas no *hardware*, auditá-las quando não o forem e ancorá-las no identificador adequado conforme a evidência do ataque é a condição que esta dissertação demonstrou ser exequível em equipamento comercial.

REFERÊNCIAS BIBLIOGRÁFICAS

1 KREUTZ, D.; RAMOS, F. M. V.; VERÍSSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, IEEE, v. 103, n. 1, p. 14–76, 2015.

2 MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008.

3 SCOTT-HAYWARD, S.; NATARAJAN, S.; SEZER, S. A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, IEEE, v. 18, n. 1, p. 623–654, 2016.

4 ANTONAKAKIS, M.; APRIL, T.; BAILEY, M.; BERNHARD, M.; BURSZTEIN, E.; COCHRAN, J.; DURUMERIC, Z.; HALDERMAN, J. A.; INVERNIZZI, L.; KALLITSIS, M.; KUMAR, D.; LEVER, C.; MA, Z.; MASON, J.; MENSCHER, D.; SEAMAN, C.; SULLIVAN, N.; THOMAS, K.; ZHOU, Y. Understanding the Mirai botnet. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017. p. 1093–1110.

5 Fortinet. *Global Threat Landscape Report 2H 2023*. Sunnyvale, CA, USA, 2024. Disponível em: <https://www.fortinet.com/content/dam/fortinet/assets/threat-reports/threat-landscape-report-2h-2023.pdf>. Acesso em: 9 jul. 2026.

6 ASLAM, N.; SRIVASTAVA, S.; GORE, M. M. ONOS DDoS defender: A comparative analysis of existing DDoS attack datasets using ensemble approach. *Wireless Personal Communications*, Springer, v. 133, n. 3, p. 1805–1827, 2024.

7 SHAIK, K. M. N. SDN-based detection and mitigation of botnet traffic in large-scale networks. *World Journal of Advanced Research and Reviews*, v. 25, n. 02, p. 2773–2784, 2025.

8 SHINAN, K.; ALSUBHI, K.; ALZAHRANI, A.; ASHRAF, M. U. Machine learning-based botnet detection in software-defined network: A systematic review. *Symmetry*, MDPI, v. 13, n. 5, p. 866, 2021.

9 WIJERATNE, S.; EKANAYAKE, A.; JAYAWEERA, S.; RAVISHAN, D.; PASQUAL, A. Scalable high performance SDN switch architecture on FPGA for core networks. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Seaside, CA, USA: ACM, 2019. p. 116.

10 TOOTOONCHIAN, A.; GORBUNOV, S.; GANJALI, Y.; CASADO, M.; SHERWOOD, R. On controller performance in software-defined networks. In: *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*. San Jose, CA, USA: USENIX Association, 2012.

11 SOUZA, C. H. M.; ARIMA, C. H. Detecção de malware em ambientes IoT habilitados por SDN. In: *Anais do XV Workshop de Pesquisa Experimental da Internet do Futuro (WPEIF)*. Porto Alegre: SBC, 2024.

12 ZARGAR, S. T.; JOSHI, J.; TIPPER, D. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys & Tutorials*, IEEE, v. 15, n. 4, p. 2046–2069, 2013.

13 GIL, A. C. *Como Elaborar Projetos de Pesquisa*. 6. ed. São Paulo: Atlas, 2019.

14 MARCONI, M. d. A.; LAKATOS, E. M. *Fundamentos de Metodologia Científica*. 9. ed. São Paulo: Atlas, 2021.

15 LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*. Monterey, CA, USA: ACM, 2010. p. 1–6.

16 SILVA, C. M.; SANTOS, V. L. d.; NZE, G. D. A.; GONDIM, J. J. C.; FILHO, F. L. d. C.; MENDONÇA, F. L. L. d. Defesa de borda orientada a SDN: Desafios de sincronização e persistência de regras na mitigação de ameaças Mirai-like. In: *Anais da 21ª Conferência Ibérica de Sistemas e Tecnologias de Informação (CISTI 2026)*. [S.l.: s.n.], 2026. Submetido para publicação.

17 SILVA, C. M.; GONDIM, J. J. C.; TORRES, J. O. A. d. L.; NZE, G. D. A.; ALBUQUERQUE, R. d. O.; MENDONÇA, F. L. L. d. Proposal for a 6G-integrated SDN security architecture for adaptive mitigation of Mirai attacks in IoT ecosystems. *Engineering Proceedings (MDPI)*, 2026. Submetido para publicação.

18 Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.5*. Palo Alto, CA, USA, 2015.

19 BENAMRANE, F.; MAMOUN, M. B.; BENAINI, R. Performances of OpenFlow-based software-defined networks: An overview. *Journal of Networks*, v. 10, n. 6, p. 329–337, 2015.

20 BERDE, P.; GEROLA, M.; HART, J.; HIGUCHI, Y.; KOBAYASHI, M.; KOIDE, T.; LANTZ, B.; O'CONNOR, B.; RADOSLAVOV, P.; SNOW, W.; PARULKAR, G. ONOS: Towards an open, distributed SDN OS. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*. Chicago, IL, USA: ACM, 2014. p. 1–6.

21 MUQADDAS, A. S.; GIACCONE, P.; BIANCO, A.; MAIER, G. Inter-controller traffic to support consistency in ONOS clusters. *IEEE Transactions on Network and Service Management*, IEEE, v. 14, n. 4, p. 1018–1031, 2017.

22 KOLIAS, C.; KAMBOURAKIS, G.; STAVROU, A.; VOAS, J. DDoS in the IoT: Mirai and other botnets. *Computer*, IEEE, v. 50, n. 7, p. 80–84, 2017.

23 MEIDAN, Y.; BOHADANA, M.; MATHOV, Y.; MIRSKY, Y.; BREITENBACHER, D.; SHABTAI, A.; ELOVICI, Y. N-BaIoT—Network-based detection of IoT botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, IEEE, v. 17, n. 3, p. 12–22, 2018.

24 BRAGA, R.; MOTA, E.; PASSITO, A. Lightweight DDoS flooding attack detection using NOX/OpenFlow. In: *Proceedings of the 35th Annual IEEE Conference on Local Computer Networks (LCN)*. Denver, CO, USA: IEEE, 2010. p. 408–415.

25 GIOTIS, K.; ARGYROPOULOS, C.; ANDROULIDAKIS, G.; KALOGERAS, D.; MAGLARIS, V. Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments. *Computer Networks*, Elsevier, v. 62, p. 122–136, 2014.

26 ZHANG, M.; LI, G.; WANG, S.; LIU, C.; CHEN, A.; HU, H.; GU, G.; LI, Q.; XU, M.; WU, J. Poseidon: Mitigating volumetric DDoS attacks with programmable switches. In: *27th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society, 2020.

27 LIU, Z.; NAMKUNG, H.; NIKOLAIDIS, G.; LEE, J.; KIM, C.; JIN, X.; BRAVERMAN, V.; YU, M.; SEKAR, V. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches. In: *30th USENIX Security Symposium (USENIX Security 21)*. Virtual Event: USENIX Association, 2021. p. 3829–3846.

28 XING, J.; WU, W.; CHEN, A. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In: *30th USENIX Security Symposium (USENIX Security 21)*. Virtual Event: USENIX Association, 2021. p. 3865–3881.

29 ILHA, A. d. S.; LAPOLLI, Â. C.; MARQUES, J. A.; GASPARY, L. P. Euclid: A fully in-network, P4-based approach for real-time DDoS attack detection and mitigation. *IEEE Transactions on Network and Service Management*, IEEE, v. 18, n. 3, p. 3121–3139, 2021.

30 ZHOU, H.; HONG, S.; LIU, Y.; LUO, X.; LI, W.; GU, G. Mew: Enabling large-scale and dynamic link-flooding defenses on programmable switches. In: *2023 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2023. p. 3178–3192.

31 YAN, J.; ZHOU, H.; WANG, W. Intelligent network element: A programmable switch based on machine learning to defend against DDoS attacks. *Information Systems Frontiers*, Springer, 2025.

32 ROMANI, M. F. et al. Mitigação de ataques DDoS em redes IoT com aprendizado federado. *Revista Ibérica de Sistemas e Tecnologias de Informação*, n. E77, p. 160–172, 2025.

33 MBONGO, K. H. R. et al. Conv1D-GRU-Self attention: An efficient deep learning framework for detecting intrusions in wireless sensor networks. *Future Internet*, MDPI, v. 17, n. 7, p. 301, 2025.

34 KALIMUTHU, V. K.; VELUMANI, R. Modeling of intrusion detection system using double adaptive weighting arithmetic optimization algorithm with deep learning on internet of things environment. *Brazilian Archives of Biology and Technology*, Instituto de Tecnologia do Paraná, v. 67, p. e24231010, 2024.

35 ADENUGA-TAIWO, O.; HEYDARI, S. S. *Security Analysis of ONOS Software-Defined Network Platform*. Oshawa, ON, Canada, 2016.

36 CAO, J.; XU, M.; LI, Q.; SUN, K.; YANG, Y. The LOFT attack: Overflowing SDN flow tables at a low rate. *IEEE/ACM Transactions on Networking*, IEEE, v. 31, n. 3, p. 1416–1431, 2023.

37 CHEN, K.-Y.; LIU, S.; XU, Y.; SIDDHRAU, I. K.; ZHOU, S.; GUO, Z.; CHAO, H. J. SDNShield: NFV-based defense framework against DDoS attacks on SDN control plane. *IEEE/ACM Transactions on Networking*, IEEE, v. 30, n. 1, p. 1–17, 2022.

38 DING, D.; SAVI, M.; PEDERZOLLI, F.; CAMPANELLA, M.; SIRACUSA, D. In-network volumetric DDoS victim identification using programmable commodity switches. *IEEE Transactions on Network and Service Management*, IEEE, v. 18, n. 2, p. 1191–1202, 2021.

39 TANG, D.; DAI, R.; ZUO, C.; CHEN, J.; LI, K.; QIN, Z. When SDN meets low-rate threats: A survey of attacks and countermeasures in programmable networks. *ACM Computing Surveys*, ACM, v. 57, n. 4, p. 103, 2024.

40 Dell Inc. *Dell Networking S3048-ON Specification Sheet*. Round Rock, TX, USA, 2016.

41 Dell Inc. *Dell SmartFabric OS10 User Guide, Release 10.6.0*. Round Rock, TX, USA, 2021.

42 Dell Inc. *Lista de compatibilidade de hardware do SmartFabric OS10*. 2026. Dell Knowledge Base, Artigo 000192674. Disponível em: <https://www.dell.com/support/kbdoc/000192674>.

43 Dell Inc. *Dell OpenFlow Deployment and User Guide 4.0*. Round Rock, TX, 2017. Rev. A00. Dell Software-Defined Networking (SDN).

44 Faucet SDN Project. *Faucet SDN Controller — Hardware Support*. 2024. Documentação oficial do projeto Faucet. Disponível em: <https://faucet.nz/>. Acesso em: 31 maio 2026.

45 Faucet SDN Project. *Compatibility with Dell EMC 5224F-ON (Issue #4337)*. 2023. Repositório `faucetsdn/faucet`, GitHub. Disponível em: <https://github.com/faucetsdn/faucet/issues/4337>. Acesso em: 31 maio 2026.

46 NVIDIA Corporation. *End of Life Notification: Cumulus Linux Support Service for 1G Switches*. 2021. NVIDIA Networking, Notificação LCR-000747. Disponível em: <https://network.nvidia.com/related-docs/eol/LCR-000747.pdf>. Acesso em: 31 maio 2026.

47 Open Networking Foundation. *Trellis — Open-Source Multi-Purpose L2/L3 Leaf-Spine Fabric*. 2024. Documentação oficial do projeto Trellis. Disponível em: <https://docs.trellisfabric.org/>. Acesso em: 31 maio 2026.

48 Proxmox Server Solutions GmbH. *Proxmox Virtual Environment Reference Manual*. Viena, Áustria, 2024. Disponível em: <https://pve.proxmox.com/pve-docs/>.

49 Open Networking Foundation. *ONOS Reactive Forwarding Application (`org.onosproject.fwd`)*. 2024. Repositório `opennetworkinglab/onos`, GitHub. Disponível em: <https://github.com/opennetworkinglab/onos/tree/master/apps/fwd>. Acesso em: 4 jun. 2026.

50 MIRKOVIC, J.; REIHER, P. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*, ACM, v. 34, n. 2, p. 39–53, 2004.

51 FERGUSON, P.; SENIE, D. *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*. 2000. RFC 2827 (BCP 38), Internet Engineering Task Force (IETF). Disponível em: <https://www.rfc-editor.org/rfc/rfc2827>. Acesso em: 29 maio 2026.

APÊNDICES

A. PSEUDOCÓDIGO: DDSO V19.3.1

O orquestrador DDSO evoluiu de forma iterativa a partir do protótipo inicial v16.1, e cada versão respondeu a um desafio concreto revelado pelo testbed em hardware. A trilha de desenvolvimento percorreu a suavização EMA dos contadores com detecção de rajada, a confirmação do sucesso real da instalação de regras pelo código de status HTTP com instrumentação de latência por *timestamp* absoluto, a remoção de regras por critério real e o desbloqueio orientado pelo contador da própria regra de descarte com histerese, a correção da zona morta de detecção pela taxa efetiva, a ancoragem da regra de descarte por `ETH_SRC` com escalada automática de `IPV4_SRC` para `ETH_SRC` diante de origem forjada, a consciência de *gateway* com deduplicação pré e pós-roteamento, a recuperação da cardinalidade real do ataque distribuído e a histerese de vítima que contém o atacante residual. A versão final, v19.3.1, consolida toda a trilha, e foi a versão empregada em todos os experimentos do Capítulo 4.

O Algoritmo 1 descreve, em alto nível, o ciclo de operação em malha fechada da v19.3.1.

Algoritmo 1: DDSO v19.3.1 — Ciclo de Operação em Malha Fechada

Require: Configuração $\mathcal{C}$ (limiares, modo IPS/IDS, allowlist, *block_strategy*), cliente ONOS $\mathcal{O}$, estado anterior $\mathcal{S}_{prev}$

Ensure: Plano de dados físico consistente com o plano de controle

1: RECONCILIARORFAS($\mathcal{O}$) ▷ v19: baseline limpa, remove regras de DROP órfãs no boot
2: **while** sistema ativo **do**
— Coleta paralela de dados (1 GET /hosts por ciclo) —
3: $hosts,\ flows \leftarrow$ GETPARALLEL($\mathcal{O}$)
4: $gw\_macs \leftarrow$ GETGATEWAYMACS($hosts$) ▷ v18: MAC em multi-VLAN/multi-IP/trunk = roteador
5: $stats \leftarrow$ BUILDSTATISTICS($flows,\ hosts,\ \mathcal{S}_{prev},\ \Delta t,\ gw\_macs$) ▷ EMA, taxa crua, burst, dedup pre/pos-L3, resgate anti-churn
6: $routers \leftarrow$ GETROUTERMACS($\mathcal{O}$)
7: $dst\_stats \leftarrow$ BUILDDESTINATIONSTATS($flows,\ hosts,\ gw\_macs$) ▷ v18/v19: por destino real, gateway-aware, origens efetivas
— Loop de detecção e mitigação por origem —
8: **for** cada origem k em $stats$ **do**
9: $tipos \leftarrow$ CLASSIFICAR(k) ▷ FLOOD, GRE-FLOOD, DNS-TORTURE, SCAN, BRUTE, UDP-AMP
10: $vol \leftarrow (\{\text{FLOOD}, \text{GRE-FLOOD}, \text{UDP-AMP}\} \cap tipos \neq \emptyset)$
11: **if not** vol **then** ▷ skip de trânsito só FORA de ataque de volume
12: **if** ISTRANSITNODE(k) **or** $k.mac \in routers$ **then**

```
            continuar
        end if
    end if
    if tipos = ∅ or k ∈ ACTIVEBLOCKS or k ∈ allowlist then
        continuar
    end if
    if modo = IPS and WITHINRATELIMIT then
        ancora ← ESCOLHEANCORA(C.block_strategy, tipos, k.mac)        ▷ MAC imediato em "mac"; IP em "ip"/"auto"
        ids ← DROPATTACK(k.ip, k.mac, ancora)
        VERIFICARSYNCENTRESWITCHES(O, k)        ▷ v19: estado da regra por switch
        ACTIVEBLOCKS[k] ← {ids, tipos, t_bloq, ancora, last_drop ← nulo,
         escalado ← falso, spoof_ev ← 0}
    else if modo = IDS then
        LOGALERT(k, tipos); ACTIVEBLOCKS[k] ← {∅, tipos, t_det}
    end if
  end for
— v19: defesa do ALVO sob ataque distribuído —
  if C.block_distributed then
    DECAIHISTERESEVITIMA(C.distributed_victim_ttl)
    for cada destino d em dst_stats com not d.is_transit do
        eff ← d.effective_origins        ▷ cardinalidade real recuperada pela correlação pre/pos-L3
        quente ← (d ∈ VITIMASDISTRIBUIDAS)
        agregado ← (d.rate ≥ C.dist_min_total_pps)
        if agregado and (|eff| ≥ 2 or quente) then        ▷ histerese contém o bot residual órfão
            for cada origem o em eff não bloqueada do
                ENGAJARORIGEM(o, "FLOOD/DIST")
            end for
        end if
    end for
  end if
— Verificação de consistência em malha fechada (UNBLOCK) —
  usa_contador ← (C.unblock_by_drop_counter and modo = IPS)
  for cada k em ACTIVEBLOCKS do
    if usa_contador then
        drop_pkts ← MITIGATIONDROPPACKETS(O, k, flows)
        if drop_pkts = nulo then
            ativo ← falso
        else if k.last_drop = nulo then
```

48: $ativo \leftarrow$ **verdadeiro** ▷ linha de base: recém-bloqueado
49: **else**
50: $\delta \leftarrow drop\_pkts - k.last\_drop$
51: $ativo \leftarrow (\delta > \mathcal{C}.unblock\_residual\_threshold)$ ▷ histerese de desbloqueio
52: **end if**
53: $k.last\_drop \leftarrow drop\_pkts$
54: **else**
55: $ativo \leftarrow (stats[k].pkts\_rate > \mathcal{C}.zero\_rate\_threshold)$
56: **end if**
— Escalada IPV4_SRC → ETH_SRC sob IP spoofing —
57: **if** $usa\_contador$ **and** $\mathcal{C}.block\_strategy =$ "auto" **and** $k.ancora =$ "ip" **and not** $k.escalado$ **and** $k.last\_drop \neq$ nulo **and** $(tipos \cap \{\text{FLOOD}, \text{GRE-FLOOD}, \text{UDP-AMP}\} \neq \emptyset)$ **then**
58: $fluindo \leftarrow (stats[k].pkts\_rate \geq \mathcal{C}.flood\_min\_pkts\_rate)$ ▷ flood persiste nas stats (resolvido via MAC)
59: **if not** $ativo$ **and** $fluindo$ **then** ▷ regra por IP inerte + ataque ativo = origem forjada
60: $k.spoof\_ev \mathrel{+}= 1;\ k.zero\_count \leftarrow 0$
61: **if** $k.spoof\_ev \geq \mathcal{C}.escalation\_cycles$ **then**
62: ESCALATETOMAC(k) ▷ reancora em ETH_SRC; rótulo SPOOFED
63: **end if**
64: **continuar**
65: **else**
66: $k.spoof\_ev \leftarrow 0$
67: **end if**
68: **end if**
69: **if** $ativo$ **then**
70: $k.zero\_count \leftarrow 0$
71: **else**
72: $k.zero\_count \mathrel{+}= 1$
73: **if** $k.zero\_count \geq \mathcal{C}.mitigation\_zero\_rate\_cycles$ **then**
74: $n \leftarrow$ DELETEMITIGATIONFLOWSBYCRITERIA($k.ip,\ k.mac$) ▷ delete por critério real, não por ID
75: Remover k de ACTIVEBLOCKS
76: LOG(k, "normalizado", n regras removidas)
77: **end if**
78: **end if**
79: **end for**
80: EXPORTPROMETHEUS($stats$)
81: SAVESTATE(ACTIVEBLOCKS) ▷ persistência JSON
82: $\mathcal{S}_{prev} \leftarrow stats$

83: **aguardar** Δt

84: **end while**

O procedimento ESCALATETOMAC instala a regra de descarte por `ETH_SRC` (MAC real do *bot*) antes de remover a regra por `IPV4_SRC`, evitando janela de exposição; em seguida reancora o estado do bloqueio para que a leitura do contador e o desbloqueio passem a casar por `ETH_SRC`, reinicia a linha de base do contador e marca o rótulo SPOOFED. A condição que dispara a escalada é a inércia da regra por IP (delta de pacotes nulo) enquanto o *flood* persiste nas estatísticas de origem: como o controlador ONOS agrega o tráfego por endereço MAC na camada de enlace, ele não observa os endereços IP forjados, e a ineficácia da regra por IP torna-se a evidência confiável do *spoofing*.

O procedimento BUILDDESTINATIONSTATS introduz a consciência de *gateway*: tráfego cujo destino de camada de enlace é o MAC de um roteador inter-VLAN é classificado como trânsito L3, e nunca como vítima de ataque distribuído. Sob roteamento inter-VLAN o endereço de origem do quadro é reescrito para o do *gateway* no salto L3, o que mascara a cardinalidade real do ataque no segmento pós-roteamento. A correlação entre os segmentos pré e pós-L3, restrita por volume e por VLAN de origem, recupera o conjunto de origens efetivas do ataque. A histerese de vítima distribuída, por sua vez, marca como "quente" por um número limitado de ciclos todo alvo que sofreu ataque distribuído, e assim contém também a última origem residual isolada, cuja taxa individual fica abaixo do limiar de *flood* e que, de outro modo, sustentaria um platô de tráfego sobre a vítima.

O Algoritmo 2 detalha a construção das estatísticas por origem na v19.3.1.

Algoritmo 2: BuildStatistics — Taxa Efetiva, EMA, Burst e Resgate anti-churn (v19.3.1)

Require: $flows$, $hosts$, $\mathcal{S}_{prev}$, Δt, gw_macs, $\alpha = 0.5$
Ensure: $stats$: mapa origem $\rightarrow$ {$pkts\_rate$, $pkts\_rate\_raw$, $life\_rate$, ...}

for cada $flow$ em $flows$ **do**
  if $flow.priority = 40000$ **and** $flow.treatment = \emptyset$ **then**
    continuar ▷ pula as próprias regras de DROP
  end if
  Extrair ip_{src}, mac_{src}, ip_{dst}, $proto$, $pkts$, $bytes$, $life$
  if $mac_{src} \in gw\_macs$ **and** $ip_{src} \notin$ IPs do gateway **then**
    continuar ▷ dedup: perna pós-L3 (ETH_SRC reescrito) é ignorada
  end if
  Agregar em $stats[origem]$; acumular $life\_rate \mathrel{+}= pkts / \max(life, \Delta t)$
end for
for cada origem k em $stats$ **do**
  $\delta_{pkts} \leftarrow stats[k].pkts - \mathcal{S}_{prev}[k].pkts$
  $raw \leftarrow \delta_{pkts} \,/\, \Delta t$; $stats[k].pkts\_rate\_raw \leftarrow raw$
  if $raw \geq 1.5 \times flood\_threshold$ **then**
    $stats[k].pkts\_rate \leftarrow raw$ ▷ burst: bypassa EMA
  else
    $stats[k].pkts\_rate \leftarrow \alpha \cdot raw + (1 - \alpha) \cdot \mathcal{S}_{prev}[k].pkts\_rate$ ▷ EMA, p/ exibição
  end if
end for
— Resgate anti-churn (origens efemeras com match L4) —
for cada origem k **com** $k.flows \geq \mathcal{C}.churn\_min\_flows$ **do**
  if $k.life\_rate \geq flood\_threshold$ **and** $k.pkts\_rate\_raw < flood\_threshold$ **then**
    $k.pkts\_rate\_raw \leftarrow \max(k.pkts\_rate\_raw,\ k.life\_rate)$ ▷ delta cegou; usa packets/life
  end if
end for
— Decisão de detecção usa a taxa efetiva —
em CLASSIFY*(k)**:** $taxa\_efetiva \leftarrow \max(stats[k].pkts\_rate,\ stats[k].pkts\_rate\_raw)$
  return $stats$

B. CÓDIGO-FONTE INTEGRAL: DDSO V19.3.1

Este apêndice apresenta o código-fonte completo do orquestrador DDSO na versão final v19.3.1, empregada em todos os ensaios do Capítulo 4. O sistema partiu do protótipo experimental v16.1 e evoluiu de forma iterativa, e cada versão respondeu a um desafio concreto observado no testbed em hardware. Entre as capacidades incorporadas ao longo dessa evolução estão: a suavização EMA dos contadores de tráfego com detecção de rajada; a confirmação do sucesso real da instalação de regras pelo código de status HTTP, com instrumentação de latência por *timestamp* absoluto; a remoção de regras por critério real, e não por identificador lógico, junto ao desbloqueio orientado pelo contador da própria regra de descarte com histerese, que eliminam o ciclo de bloqueio e desbloqueio prematuro; a supressão de falsos positivos em nós de trânsito; os classificadores de *GRE Flood*, *DNS Water Torture* e *Ping Sweep*; a persistência de estado em JSON; a correção da zona morta de detecção pela taxa efetiva; a mitigação resistente a *IP spoofing* por ancoragem da regra de descarte em `ETH_SRC`, com a estratégia configurável `block_strategy` e a escalada automática de `IPV4_SRC` para `ETH_SRC` quando a regra por IP permanece inerte enquanto o *flood* persiste, condição rotulada como `SPOOFED`; a consciência de *gateway* com deduplicação do tráfego pré e pós-roteamento; a recuperação da cardinalidade real do ataque distribuído pela correlação entre os segmentos de enlace; a reconciliação de regras órfãs e a verificação de sincronização da regra entre os comutadores; e a histerese de vítima distribuída, que contém o atacante residual isolado[1].

Listagem B.1: Código-fonte completo do DDSO v19.3.1 (SDN Forensic & DDoS Mitigation Platform).

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
detector_v19_3_1.py
====================
SDN Forensic & DDoS Mitigation Platform - v19.3.1

Mudancas v19.3 sobre v19.2.0:
  - CONTENCAO DO ATACANTE RESIDUAL ORFAO (fim do "plato"). Causa: um bot Mirai
    isolado sustenta ~8000 pps, ABAIXO de flood_min_pkts_rate (10000), logo nunca
    classifica como FLOOD individual. Quando o 3o flow aparece atrasado (instalacao
    reativa escalonada / CoPP) com os dois peers JA bloqueados, sobra UMA origem
    efetiva e o caminho distribuido exigia len(eff)>=2, deixando o residual orfao.
    O alvo ficava preso em ~8000 pps por dezenas de segundos (observado nos runs
    01/03/11/12 do vetor SYN). Correcao: HISTERESE DE VITIMA DISTRIBUIDA. Um alvo
    que sofreu ataque distribuido (len>=2 acima do limiar) e' marcado vitima por
    distributed_victim_ttl ciclos (renovado enquanto o residual seguir acima do
    limiar); so entao o caminho distribuido engaja UMA origem residual sozinha.
    Preserva a semantica de "distribuido" e NAO bloqueia fonte legitima isolada
    (uma fonte legitima nunca foi distribuida, logo nunca fica "quente"). Sitios:
    MitigationManager.step (bloco block_distributed) e __init__ (_dist_victims).
  - LIMIAR DO AGREGADO/RECUPERACAO = dest_distributed_min_total_pps. O gatilho do
```

[1]Para assegurar a fidelidade de impressão e a compilação em `pdflatex`, os glifos não-ASCII do painel de monitoração (emojis, blocos da *sparkline* e caracteres de moldura) foram transliterados para equivalentes ASCII na listagem; a lógica do programa é idêntica à do código executável.

```
    agregado distribuido (step) e a recuperacao de origens efetivas
    (build_destination_statistics) passam de flood_min_pkts_rate (10000) para
    dest_distributed_min_total_pps (5000), para alcancar o residual de ~8000 pps.
  - ESTIMADOR DE TAXA RESISTENTE A CHURN (SYN multi-porta efemera). Causa: a taxa
    e' medida pelo DELTA do contador acumulado entre polls. Sob portas de origem
    efemeras com match L4 e idle curto, os flows churnam, a soma por origem fica
    parada e o delta da' zero (flood invisivel). Correcao: estima a taxa tambem por
    packets/life de cada flow e usa-a SO quando ha muitos flows (churn_min_flows) e
    o delta cegou. INERTE no caso persistente (1 flow/bot, padrao do ONOS com
    matchTcpUdpPorts=false): nao altera os numeros validados (SYN 3.60s, etc.).
    Requer o campo "life" no flow stats (o ONOS expoe). Sitios: extract_flow_data,
    build_statistics (acumulo life_* e resgate por churn), OriginStats.
  - TOGGLES A/B (default = comportamento v19.3.0): residual_hysteresis e
    churn_rescue. Desligados, reproduzem o comportamento v19.2.0 (residual preso /
    churn invisivel) para o A/B da dissertacao. distributed_victim_ttl e
    churn_min_flows ajustam a janela da histerese e o gatilho do churn.

  v19.3.1 (sobre v19.3.0): SEMENTE DA HISTERESE pelos bloqueios INDIVIDUAIS. No
  ambiente real, sob chegada escalonada (reactive cold-start), os bots sao
  derrubados um a um pelo laco por-origem (cada um deu pico > flood_min_pkts_rate
  no burst do SYN) ANTES de qualquer ciclo ver 2 origens efetivas simultaneas no
  alvo (o alvo fica preso em 1 origem efetiva). Com a histerese latcheando so a
  partir de len(eff)>=2, o residual que chegava por ultimo e sozinho so era contido
  quando a correlacao por acaso revelava uma 2a origem (medido ~35s de exposicao no
  live). Agora, quando ha bloqueio volumetrico no ciclo, marca-se "quente" todo
  alvo real cujo agregado >= dest_distributed_min_total_pps; a janela e' renovada
  enquanto o residual seguir acima do limiar, entao o bot atrasado e' contido no 1o
  ciclo em que aparece (~1-2s em vez de ~35s). Gateado por residual_hysteresis
  (mesmo toggle do A/B). Sitio: MitigationManager.step.

Mudancas v19.2 sobre v19.1.0 (BUILD FINAL):
  - OTIMIZACAO DO CICLO (1 GET /hosts por ciclo): cinco funcoes derivam de
    /hosts (get_hosts, get_host_info, get_router_macs, get_gateway_macs,
    get_gateway_faces). Antes, quando os caches de 60s expiravam no mesmo
    ciclo, ate 4 GETs /hosts redundantes caiam juntos (soluco periodico de
    latencia de ciclo). Agora um cache de payload por ciclo (clear_cycle_cache
    no inicio do one_cycle, _hosts_data com lock) garante UM unico GET /hosts
    por iteracao, reaproveitado por todos os consumidores. A LOGICA de cada
    funcao e' identica (cada uma processa o mesmo payload do seu jeito); muda
    apenas a fonte do dado. NAO altera deteccao nem mitigacao: o sinal de
    ataque vem de /flows (inalterado) e o bloqueio via push_flow direto. E'
    higiene de engenharia (ciclo mais leve, sem o pico de 60s), nao latencia
    de mitigacao. Equivalencia funcional verificada por bateria de testes.
  - Nenhuma outra mudanca sobre o v19.1.0 (que ja trazia o fix do lag de
    deteccao). Todos os recursos do v19.0.0-final preservados.

Mudancas v19.1 sobre v19.0.0-final (VERSAO DE HOMOLOGACAO):
  - FIX DO LAG DE DETECCAO (~20s -> ~1 ciclo): o tratamento de "first_seen"
    introduzido no v18/v19 zerava a taxa de QUALQUER origem nova em seu
    primeiro ciclo. Isso era correto apenas no BOOT do detector (onde o
    contador acumulado do flow seria lido como taxa, gerando FLOOD falso em
    tudo), mas durante a operacao descartava o 1o pico de um flood novo e
    jogava a deteccao no caminho EMA (lento). Somado ao arranque frio do
    contador do switch, observava-se ~20s ate o bloqueio. Agora o zero so se
    aplica no PRIMEIRO ciclo de vida do detector (previous vazio); apos o boot,
    origem nova com salto vira taxa na hora e dispara o burst no 1o ciclo
    (comportamento do v17.11, validado em ambiente sao). Corrigido em
    build_statistics e build_destination_statistics. Boot-fix preservado.
  - Nenhuma outra mudanca: escalada IP->MAC, dedup pre/pos-L3, gateway-aware,
    nomes amigaveis, reconciliacao de regra orfa e verificacao de sync entre
```

```
    switches seguem identicas ao v19.0.0-final.

Mudancas v18 sobre v17.11:
  - CORRECAO DA AGREGACAO POR DESTINO (build_destination_statistics): a funcao
    agora conta por flow, direto da lista de flows, em vez de dividir a taxa do
    origem igualmente entre seus destinos (bug que diluia/distorcia o pps por
    alvo). Cada flow contribui com seu proprio contador de pacotes ao destino
    real. A taxa por destino vem do DELTA do contador acumulado do destino
    (paralelo ao build_statistics), com suavizacao EMA.
  - GATEWAY-AWARE: trafego cujo destino L2 e o MAC de um gateway/roteador e
    TRANSITO L3, nao vitima. Marcado com is_transit=True e nunca classificado
    como DDoS distribuido. Deteccao de gateway deterministica (get_gateway_macs):
    MAC presente em mais de uma VLAN OU com mais de um IP OU em trunk (vlan=None).
    Elimina o next(iter(set)) nao-deterministico da resolucao MAC->IP.
    Fundamento honesto: na topologia roteada com Reactive Forwarding em L2, o IP
    L3 final de trafego PRE-roteado NAO e observavel pelo controlador (o flow
    casa ETH_SRC/ETH_DST, sem IPV4_*). O alvo aparece pelo segmento POS-L3
    (ETH_DST = MAC do alvo), que reflete o que de fato chegou ao alvo.
  - NOMES AMIGAVEIS (hostname): origens, bloqueios, monitoracao, alvos e logs de
    acao passam a exibir "nome (IP)" na mesma linha, resolvendo o friendly name
    a partir das annotations dos hosts no ONOS (h.annotations.name). Cai de volta
    para IP/MAC quando o nome nao existe.

Mudancas v17.11 sobre v17.10:
  - ROTULAGEM FIEL DE SPOOFING (modo "auto"): o "auto" agora bloqueia primeiro
    por IPV4_SRC e, no loop de consistencia, observa o contador dessa regra. Se a
    regra por IP fica inerte (delta ~0) ENQUANTO o flood persiste nas stats
    (origem ainda em alta taxa, resolvida via MAC), conclui-se que a origem-IP e
    volatil/forjada -- pois o ONOS agrega por MAC na L2 e o controlador nao ve os
    IPs falsos. Nesse caso ESCALA para bloqueio por ETH_SRC e marca o rotulo
    FLOOD/SPOOFED. Floods de origem fixa nunca sao rotulados SPOOFED (a regra por
    IP captura os pacotes; o contador sobe). Os modos "mac" (bloqueio por MAC
    imediato) e "ip" (legado, sem escalada) seguem inalterados.
  - Novo campo escalation_cycles (ciclos de evidencia antes de escalar; default 1).
  - Novo log [ESCALADA] e flag de estado spoofed por bloqueio.

Mudancas v17.10 sobre v17.9.1:
  - MITIGACAO RESISTENTE A IP SPOOFING: nova estrategia de ancoragem da regra
    de DROP (block_strategy). Sob spoofing de origem, o IPV4_SRC e volatil e a
    regra por IP nao casa os pacotes do ataque; o ETH_SRC (MAC real do bot)
    permanece estavel na borda L2 e casa TODOS os pacotes, IP forjado ou nao.
    Em "auto" (default), floods volumetricos (FLOOD/GRE-FLOOD/UDP-AMP) passam a
    ser bloqueados por ETH_SRC; SCAN/BRUTE seguem por IPV4_SRC. Modos "ip" e
    "mac" forcam a ancora (uteis para o A/B na dissertacao). O UNBLOCK por
    contador e o delete tambem casam por ETH_SRC quando a regra e ancorada em
    MAC (entrada E liberacao). Caveats: so funciona com o MAC observavel no
    mesmo dominio L2 (borda); corta TODO o trafego daquele host.
  - Flag de CLI --block-strategy {ip,mac,auto}.

Mudancas v17.9 sobre v17.8:
  - FIX da ZONA MORTA DE DETECCAO: a decisao de classificacao (classify_flood,
    classify_gre_flood, classify_amplification, classify_dns_water_torture)
    passa a usar a taxa EFETIVA = max(taxa_crua, taxa_suavizada) em vez da
    taxa suavizada por EMA.
  - Diagnostico instrumentado: log DEBUG "DIAG rate" por ciclo (ative com --debug).

Mudancas v17.8 sobre v17.7:
  - Novo painel "Monitoracao de Bloqueios" no dashboard.
  - v17.8.1: suavizacao EMA da taxa EXIBIDA no painel.
```

```
Mudancas v17.7 sobre v17.6:
  - FIX do UNBLOCK PREMATURO: o desbloqueio observa o DELTA de pacotes da
    PROPRIA regra de DROP no switch (via ONOS) em vez do trafego "vazado".
  - v17.7.1: HISTERESE no UNBLOCK (unblock_residual_threshold, default 100).

Mudancas v17.6 sobre v17.5:
  - Instrumentacao de latencia por timestamp (epoch absoluto).
  - push_flow confirma o sucesso real via status_code.
  - burst detection passa a usar cfg.burst_multiplier e cfg.flood_min_pkts_rate.

Uso:
    python3 detector_v19_3_1.py [--config config.yaml] [--mode ips|ids]
                                [--block-strategy ip|mac|auto] [--theme dark|light]
"""
from __future__ import annotations

import argparse
import csv
import datetime
import json
import logging
import os
import statistics
import sys
import time
import traceback
from collections import Counter, defaultdict, deque
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any, Deque, Dict, List, Optional, Set, Tuple

import concurrent.futures
import requests

# --- Optional deps ---
try:
    import yaml  # type: ignore
    YAML_ENABLED = True
except ImportError:
    YAML_ENABLED = False

try:
    from prometheus_client import Gauge, start_http_server
    PROMETHEUS_ENABLED = True
except ImportError:
    PROMETHEUS_ENABLED = False

try:
    from rich.console import Console
    from rich.layout import Layout
    from rich.live import Live
    from rich.panel import Panel
    from rich.table import Table
    from rich.text import Text
    RICH_ENABLED = True
except ImportError:
    RICH_ENABLED = False


# ============================================================================
```

```
# 1. CONFIGURACAO
# ============================================================================
@dataclass
class Config:
    # ONOS
    onos_url: str = "http://127.0.0.1:8181/onos/v1"
    onos_user: str = "onos"
    onos_pass: str = "rocks"
    onos_timeout: float = 5.0

    # Operacao
    mode: str = "ids"  # "ips" ou "ids"
    interval: float = 1.0
    prometheus_port: int = 8000
    csv_filename: str = "mitigacao_forense_log.csv"
    state_filename: str = "detector_state.json"
    log_filename: str = "detector.log"

    # Heuristicas
    scan_min_destinations: int = 5
    scan_min_flows: int = 5
    scan_max_pkts_avg: int = 200
    brute_min_conns: int = 20
    brute_ports: Set[int] = field(default_factory=lambda: {22, 23, 2323, 80, 8080, 7547,
    ↪ 37215})
    flood_min_pkts_rate: int = 10_000
    flood_min_bytes_rate: int = 10_000_000
    flood_max_destinations: int = 3
    amp_ports: Set[int] = field(default_factory=lambda: {53, 123, 1900})
    amp_min_avg_packet_size: int = 500
    amp_min_pkts_rate: int = 100  # evita falso positivo em DNS legitimo

    # DNS water torture (Mirai ATK_VEC_DNS)
    dns_torture_min_queries: int = 500
    dns_torture_min_rate: int = 100

    # GRE flood (Mirai ATK_VEC_GREIP/GREETH, IP_PROTO=47)
    gre_min_pkts_rate: int = 5_000

    # Agregacao por destino (detecta ataque distribuido)
    dest_distributed_min_origins: int = 3
    dest_distributed_min_total_pps: int = 5_000

    # Mitigacao
    mitigation_zero_rate_cycles: int = 3
    ids_alert_clear_cycles: int = 5
    max_mitigations_per_minute: int = 20
    zero_rate_threshold: float = 5.0
    allowlist_ips: Set[str] = field(default_factory=set)
    allowlist_macs: Set[str] = field(default_factory=set)

    # UNBLOCK orientado pelo contador da regra de DROP.
    unblock_by_drop_counter: bool = True
    unblock_residual_threshold: int = 100

    # Dashboard
    history_size: int = 60
    dashboard_top_n: int = 8
    theme: str = "dark"
    menu_key: str = "m"
```

```
    show_friendly: bool = True  # v18: exibir "nome (IP)" quando houver friendly name
    reconcile_on_boot: bool = True  # v19beta: remove regras de DROP orfas no startup
    sync_verify: bool = True        # v19beta: verifica a regra por switch apos o bloqueio

    # Deteccao de nos de transito (anti-falso-positivo)
    transit_detection_enabled: bool = True
    transit_threshold_pps: float = 1000.0
    auto_detect_routers: bool = True
    topology_cache_ttl: float = 60.0
    skip_icmp_scan: bool = True
    burst_detection_enabled: bool = True
    burst_multiplier: float = 1.5

    # Ancora da regra de DROP (resistencia a IP spoofing)
    block_strategy: str = "auto"
    volumetric_labels: Set[str] = field(
        default_factory=lambda: {"FLOOD", "GRE-FLOOD", "UDP-AMP"}
    )
    escalation_cycles: int = 1

    # v19fix: AGIR em ataque DDoS DISTRIBUIDO.
    block_distributed: bool = True

    # v19.3: HISTERESE DE VITIMA DISTRIBUIDA.
    residual_hysteresis: bool = True
    distributed_victim_ttl: int = 10

    # v19.3: ESTIMADOR RESISTENTE A CHURN.
    churn_rescue: bool = True
    churn_min_flows: int = 4

    @classmethod
    def load(cls, yaml_path: Optional[str] = None) -> "Config":
        cfg = cls()
        if yaml_path and YAML_ENABLED and Path(yaml_path).exists():
            with open(yaml_path) as f:
                data = yaml.safe_load(f) or {}
            for k, v in data.items():
                if hasattr(cfg, k):
                    if isinstance(getattr(cfg, k), set) and isinstance(v, list):
                        v = set(v)
                    setattr(cfg, k, v)
        cfg.onos_url = os.getenv("ONOS_URL", cfg.onos_url)
        cfg.onos_user = os.getenv("ONOS_USER", cfg.onos_user)
        cfg.onos_pass = os.getenv("ONOS_PASS", cfg.onos_pass)
        return cfg


# ============================================================================
# 2. LOGGING
# ============================================================================
def setup_logging(log_file: str, level: int = logging.INFO) -> logging.Logger:
    logger = logging.getLogger("detector")
    logger.setLevel(level)
    logger.handlers.clear()

    fmt = logging.Formatter(
        "%(asctime)s.%(msecs)03d [%(levelname)s] %(name)s: %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
    )
```

```
322
323     fh = logging.FileHandler(log_file)
324     fh.setFormatter(fmt)
325     logger.addHandler(fh)
326
327     sh = logging.StreamHandler(sys.stderr)
328     sh.setLevel(logging.WARNING)
329     sh.setFormatter(fmt)
330     logger.addHandler(sh)
331
332     return logger
333
334
335 log = logging.getLogger("detector")
336
337
338 # ============================================================================
339 # 3. UTILS
340 # ============================================================================
341 SPARK_CHARS = "\u2581\u2582\u2583\u2584\u2585\u2586\u2587\u2588"
342
343
344 def sparkline(values: List[float], width: int = 20) -> str:
345     if not values:
346         return " " * width
347     vals = list(values)[-width:]
348     mx = max(vals) if vals else 0
349     if mx == 0:
350         return SPARK_CHARS[0] * len(vals)
351     n = len(SPARK_CHARS) - 1
352     return "".join(SPARK_CHARS[min(int(v / mx * n), n)] for v in vals)
353
354
355 def fmt_label(name: Optional[str], primary: Optional[str], show: bool = True) -> str:
356     """Formata a identidade para exibicao. Com friendly name: 'nome (primary)'.
357     Sem nome: so o primary (IP ou MAC). primary nunca deve ser None aqui."""
358     primary = primary or "\u2014"
359     if show and name and name != primary:
360         return f"{name} ({primary})"
361     return primary
362
363
364 OUI_VENDORS: Dict[str, str] = {
365     "BC:24:11": "Proxmox/QEMU",
366     "52:54:00": "QEMU virt",
367     "B8:27:EB": "Raspberry Pi",
368     "DC:A6:32": "Raspberry Pi 4",
369     "E4:5F:01": "Raspberry Pi",
370     "00:0C:29": "VMware",
371     "00:50:56": "VMware",
372     "08:00:27": "VirtualBox",
373     "02:42:AC": "Docker bridge",
374     "00:15:5D": "Hyper-V",
375     "64:1C:67": "Cisco",
376     "00:01:42": "Cisco",
377     "F4:CF:E2": "Cisco",
378     "00:1B:21": "Intel",
379     "00:E0:4C": "Realtek",
380     "00:1D:09": "Dell",
381     "00:0E:08": "Cisco-Linksys",
```

```
    "F8:75:88": "Mikrotik",
    "E4:8D:8C": "Routerboard/Mikrotik",
    "94:DE:80": "Giga-Byte",
}


def mac_vendor(mac: Optional[str]) -> Optional[str]:
    if not mac or len(mac) < 8:
        return None
    oui = mac[:8].upper()
    return OUI_VENDORS.get(oui)


def is_suspicious_subnet(ip: Optional[str], expected_subnets: Set[str]) -> bool:
    if not ip or not expected_subnets:
        return False
    parts = ip.split(".")
    if len(parts) < 3:
        return True
    prefix = ".".join(parts[:3])
    return prefix not in expected_subnets


# ============================================================================
# 4. DATA MODELS
# ============================================================================
@dataclass
class OriginStats:
    flows: int = 0
    pkts: int = 0
    bytes: int = 0
    destinations: Set[str] = field(default_factory=set)
    per_flow_pkts: List[int] = field(default_factory=list)
    dst_port_counter: Counter = field(default_factory=Counter)
    udp_src_counter: Counter = field(default_factory=Counter)
    ip_protos: Set[int] = field(default_factory=set)
    ip_src: Optional[str] = None
    mac_src: Optional[str] = None
    pkts_rate: float = 0.0
    bytes_rate: float = 0.0
    pkts_rate_raw: float = 0.0
    bytes_rate_raw: float = 0.0
    life_pkts_rate: float = 0.0   # v19.3: taxa estimada por packets/life (resistente a churn)
    life_bytes_rate: float = 0.0  # v19.3: idem para bytes
    vlan: Optional[str] = None
    vendor: Optional[str] = None
    friendly_name: Optional[str] = None  # v18: hostname amigavel (ONOS annotations)
    l3_rewritten: bool = False  # v19beta-fix: este IP tambem teve perna pos-L3 (ETH_SRC reescrito p/ gateway), pulada na dedup


@dataclass
class DestinationStats:
    """Visao agregada por destino (detecta ataque distribuido)."""
    dst: str
    origins: Set[str] = field(default_factory=set)
    total_pkts: int = 0
    total_pkts_rate: float = 0.0
    total_bytes_rate: float = 0.0
```

```
    is_transit: bool = False              # v18: salto de roteamento, nao vitima
    friendly_name: Optional[str] = None   # v18: hostname amigavel do destino
    vlan: Optional[str] = None            # v19beta: VLAN do destino (coluna nova)
    display_ip: Optional[str] = None      # v19beta: IP de exibicao (resolve gateway)
    effective_origins: Set[str] = field(default_factory=set)


@dataclass
class BlockState:
    flow_ids: List[str]
    zero_rate_count: int
    attack_label: str
    blocked_at: float
    target_ip: Optional[str] = None
    target_mac: Optional[str] = None
    last_drop_pkts: Optional[int] = None
    drop_pkts_total: int = 0
    drop_pps: float = 0.0
    drop_pps_ema: float = 0.0
    last_active: bool = True
    anchor: str = "ip"
    escalated: bool = False
    spoof_evidence: int = 0
    spoofed: bool = False
    friendly_name: Optional[str] = None   # v18: hostname do alvo bloqueado
    observed_pps: float = 0.0             # v19beta: taxa observada da origem (IDS)


@dataclass
class TargetImpact:
    """v19beta - Separa o que foi ENTREGUE a alvos reais (pos-L3) do que esta em
    TRANSITO rumo ao gateway."""
    per_target: Dict[str, Dict[str, float]] = field(default_factory=dict)
    transit_pkts: int = 0
    transit_rate: float = 0.0
    efficacy_estimate: Optional[float] = None


@dataclass
class SyncReport:
    """v19beta - Convergencia da regra de DROP entre switches."""
    devices: List[str] = field(default_factory=list)
    added_at: Dict[str, float] = field(default_factory=dict)
    converged: bool = False


# ============================================================================
# 5. ONOS CLIENT
# ============================================================================
class OnosClient:
    """Cliente ONOS com cache de device_ids e pool de threads reutilizado."""

    def __init__(self, cfg: Config):
        self.cfg = cfg
        self._device_cache: Tuple[float, List[str]] = (0.0, [])
        self._executor = concurrent.futures.ThreadPoolExecutor(max_workers=8)
        self.last_call_ms: float = 0.0
        self.last_call_ok: bool = True
        self._friendly: Dict[str, str] = {}  # v18: mapa mac/ip -> friendly name
        self.call_log: List[Tuple[str, float, int]] = []
```

```
500         self._hosts_cycle: Optional[dict] = None
501         self._hosts_lock = __import__("threading").Lock()
502 
503     def clear_cycle_cache(self) -> None:
504         """v19.2: invalida o payload de /hosts no inicio de cada ciclo."""
505         with self._hosts_lock:
506             self._hosts_cycle = None
507 
508     def _hosts_data(self) -> dict:
509         """v19.2: retorna o payload de /hosts buscando no MAXIMO uma vez por
510         ciclo."""
511         with self._hosts_lock:
512             if self._hosts_cycle is not None:
513                 return self._hosts_cycle
514             data = self._get("hosts")
515             self._hosts_cycle = data
516             return data
517 
518     def _get(self, endpoint: str, params: Optional[dict] = None) -> dict:
519         url = f"{self.cfg.onos_url}/{endpoint.lstrip('/')}"
520         t0 = time.perf_counter()
521         try:
522             r = requests.get(
523                 url,
524                 auth=(self.cfg.onos_user, self.cfg.onos_pass),
525                 params=params,
526                 timeout=self.cfg.onos_timeout,
527             )
528             r.raise_for_status()
529             data = r.json()
530             self.last_call_ms = (time.perf_counter() - t0) * 1000
531             self.last_call_ok = True
532             n = 0
533             if isinstance(data, dict):
534                 for v in data.values():
535                     if isinstance(v, list):
536                         n = len(v)
537                         break
538             self.call_log.append((endpoint, self.last_call_ms, n))
539             return data
540         except requests.exceptions.Timeout:
541             log.warning("ONOS timeout em %s", endpoint)
542         except requests.exceptions.HTTPError as e:
543             log.error("ONOS HTTP %s em %s", e.response.status_code, endpoint)
544         except Exception:
545             log.exception("Falha em GET %s", endpoint)
546         self.last_call_ms = (time.perf_counter() - t0) * 1000
547         self.last_call_ok = False
548         self.call_log.append((endpoint, self.last_call_ms, -1))  # instrum: -1 = falha/
    ↪ timeout
549         return {}
550 
551     def get_device_ids(self, ttl: float = 30.0) -> List[str]:
552         ts, cached = self._device_cache
553         if time.time() - ts < ttl and cached:
554             return cached
555         data = self._get("devices").get("devices", [])
556         ids = [
557             d.get("id")
558             for d in data
```

```
            if d.get("available") and (d.get("id") or "").startswith("of:")
        ]
        self._device_cache = (time.time(), ids)
        return ids

    @staticmethod
    def _host_name(h: dict) -> Optional[str]:
        """v18: extrai o friendly name de um host ONOS."""
        ann = h.get("annotations") or {}
        return ann.get("name") or h.get("name")

    def get_hosts(self) -> Dict[str, Dict[str, Any]]:
        info: Dict[str, Dict[str, Any]] = defaultdict(
            lambda: {"ips": set(), "vlan": None, "locations": [], "name": None}
        )
        friendly: Dict[str, str] = {}
        data = self._hosts_data()
        for h in data.get("hosts", []):
            mac = h.get("mac")
            if not mac:
                continue
            name = self._host_name(h)
            for ip in h.get("ipAddresses", []):
                info[mac]["ips"].add(ip)
                if name:
                    friendly[ip] = name
            vlan = h.get("vlan")
            if vlan is not None and str(vlan).lower() != "none":
                info[mac]["vlan"] = vlan
            info[mac]["locations"] = h.get("locations", [])
            if name:
                info[mac]["name"] = name
                friendly[mac] = name
        self._friendly = friendly  # v18: disponibiliza para o ciclo
        return info

    def get_friendly_names(self) -> Dict[str, str]:
        """v18: mapa {mac->nome, ip->nome} populado pelo ultimo get_hosts()."""
        return self._friendly

    def get_flows(self, device_id: Optional[str] = None) -> List[dict]:
        params = {"deviceId": device_id} if device_id else None
        return self._get("flows", params=params).get("flows", [])

    def get_hosts_and_flows_parallel(
        self, device_id: Optional[str] = None
    ) -> Tuple[Dict[str, Dict[str, Any]], List[dict]]:
        f_hosts = self._executor.submit(self.get_hosts)
        f_flows = self._executor.submit(self.get_flows, device_id)
        return f_hosts.result(), f_flows.result()

    def push_flow(self, device_id: str, payload: dict) -> Optional[str]:
        url = f"{self.cfg.onos_url}/flows/{device_id}"
        try:
            r = requests.post(
                url,
                auth=(self.cfg.onos_user, self.cfg.onos_pass),
                json={**payload, "deviceId": device_id},
                timeout=2,
            )
```

```
            if r.status_code in (200, 201, 204):
                return payload.get("id")
            log.error("push_flow device=%s status=%s", device_id, r.status_code)
            return None
        except Exception:
            log.exception("Falha em push_flow device=%s", device_id)
            return None

    def delete_flow(self, device_id: str, flow_id: str) -> bool:
        url = f"{self.cfg.onos_url}/flows/{device_id}/{flow_id}"
        try:
            r = requests.delete(
                url, auth=(self.cfg.onos_user, self.cfg.onos_pass), timeout=2
            )
            return r.status_code in (204, 404)
        except Exception:
            log.exception("Falha em delete_flow device=%s flow=%s", device_id, flow_id)
            return False

    def push_flow_to_all(self, payload: dict) -> List[str]:
        device_ids = self.get_device_ids()
        if not device_ids:
            return []
        injected: Set[str] = set()
        futures = [
            self._executor.submit(self.push_flow, did, dict(payload))
            for did in device_ids
        ]
        for f in concurrent.futures.as_completed(futures):
            r = f.result()
            if r:
                injected.add(r)
        return list(injected)

    def delete_flows_from_all(self, flow_ids: List[str]) -> bool:
        if not flow_ids:
            return False
        device_ids = self.get_device_ids()
        if not device_ids:
            return False
        ok = True
        for flow_id in flow_ids:
            futures = [
                self._executor.submit(self.delete_flow, did, flow_id)
                for did in device_ids
            ]
            for f in concurrent.futures.as_completed(futures):
                if not f.result():
                    ok = False
        return ok

    def get_host_info(self, ttl: float = 60.0) -> Dict[str, Dict[str, Any]]:
        now = time.time()
        if hasattr(self, "_host_info_cache") and now - self._host_info_cache[0] < ttl:
            return self._host_info_cache[1]
        info: Dict[str, Dict[str, Any]] = {}
        data = self._hosts_data()
        for h in data.get("hosts", []):
            mac = h.get("mac")
            if not mac:
```

```
                continue
            info[mac] = {
                "vlan": h.get("vlan"),
                "ips": set(h.get("ipAddresses", [])),
                "locations": h.get("locations", []),
                "name": self._host_name(h),
            }
        self._host_info_cache = (now, info)
        return info

    def get_edge_locations(self, ttl: float = 60.0) -> Set[Tuple[str, str]]:
        edges: Set[Tuple[str, str]] = set()
        for mac, meta in self.get_host_info(ttl).items():
            for loc in meta.get("locations", []):
                dev = loc.get("elementId")
                port = str(loc.get("port", ""))
                if dev and port:
                    edges.add((dev, port))
        return edges

    def get_router_macs(self, ttl: float = 60.0) -> Set[str]:
        """Identifica MACs de routers via heuristica vlan=None."""
        now = time.time()
        if hasattr(self, "_router_cache") and now - self._router_cache[0] < ttl:
            return self._router_cache[1]
        macs: Set[str] = set()
        data = self._hosts_data()
        for h in data.get("hosts", []):
            mac = h.get("mac")
            if not mac:
                continue
            vlan = h.get("vlan")
            if vlan in (None, "None", "none"):
                macs.add(mac)
        self._router_cache = (now, macs)
        return macs

    # v18 - deteccao deterministica de MACs de gateway/roteador.
    def get_gateway_macs(self, ttl: float = 60.0) -> Set[str]:
        """Um MAC que aparece em MAIS DE UMA VLAN, OU com MAIS DE UM IP, OU em
        trunk (vlan=None) e', neste testbed, o roteador inter-VLAN."""
        now = time.time()
        if hasattr(self, "_gw_cache") and now - self._gw_cache[0] < ttl:
            return self._gw_cache[1]
        by_mac_vlans: Dict[str, Set[Any]] = defaultdict(set)
        by_mac_ips: Dict[str, Set[str]] = defaultdict(set)
        data = self._hosts_data()
        for h in data.get("hosts", []):
            mac = h.get("mac")
            if not mac:
                continue
            by_mac_vlans[mac].add(h.get("vlan"))
            for ip in h.get("ipAddresses", []):
                by_mac_ips[mac].add(ip)
        gw: Set[str] = set()
        for mac, vlans in by_mac_vlans.items():
            multi_vlan = len(vlans) > 1
            multi_ip = len(by_mac_ips[mac]) > 1
            in_trunk = any(v in (None, "None", "none") for v in vlans)
            if multi_vlan or multi_ip or in_trunk:
```

```
                gw.add(mac)
        self._gw_cache = (now, gw)
        return gw

    # v19beta - faces do gateway por VLAN.
    def get_gateway_faces(self, ttl: float = 60.0) -> Dict[str, Dict[str, Dict[str,
    ↪ Optional[str]]]]:
        now = time.time()
        if hasattr(self, "_gwfaces_cache") and now - self._gwfaces_cache[0] < ttl:
            return self._gwfaces_cache[1]
        faces: Dict[str, Dict[str, Dict[str, Optional[str]]]] = defaultdict(dict)
        data = self._hosts_data()
        for h in data.get("hosts", []):
            mac = h.get("mac")
            if not mac:
                continue
            vlan = str(h.get("vlan"))
            ips = sorted(h.get("ipAddresses", []))
            faces[mac][vlan] = {
                "name": self._host_name(h),
                "ip": ips[0] if ips else None,
            }
        out = {k: dict(v) for k, v in faces.items()}
        self._gwfaces_cache = (now, out)
        return out

    def delete_mitigation_flows_for(
        self, target_ip: Optional[str], target_mac: Optional[str]
    ) -> int:
        """Apaga TODAS as flows priority=40000 cujo IPV4_SRC ou ETH_SRC case com o
        target_ip/target_mac. Retorna quantas deletou."""
        flows = self.get_flows()
        deleted = 0
        for f in flows:
            if f.get("priority") != 40000:
                continue
            instructions = f.get("treatment", {}).get("instructions", [])
            is_drop = (not instructions) or all(
                i.get("type") == "NOACTION" for i in instructions
            )
            if not is_drop:
                continue  # v18: protege LLDP/BDDP/ARP (treatment OUTPUT)
            crits = f.get("selector", {}).get("criteria", [])
            matched = False
            for c in crits:
                t = c.get("type")
                if t == "IPV4_SRC" and target_ip:
                    ip = (c.get("ip") or "").split("/")[0]
                    if ip == target_ip:
                        matched = True
                        break
                if t == "ETH_SRC" and target_mac:
                    if c.get("mac") == target_mac:
                        matched = True
                        break
            if matched:
                dev = f.get("deviceId")
                fid = f.get("id")
                if dev and fid and self.delete_flow(dev, fid):
                    deleted += 1
```

```
                    log.info("Flow %s deletada do device %s", fid, dev)
        return deleted

    def mitigation_flow_state(
        self, target_ip: Optional[str], target_mac: Optional[str]
    ) -> Optional[str]:
        for f in self.get_flows():
            if f.get("priority") != 40000:
                continue
            instructions = f.get("treatment", {}).get("instructions", [])
            is_drop = (not instructions) or all(
                i.get("type") == "NOACTION" for i in instructions
            )
            if not is_drop:
                continue
            for c in f.get("selector", {}).get("criteria", []):
                t = c.get("type")
                if t == "IPV4_SRC" and target_ip:
                    if (c.get("ip") or "").split("/")[0] == target_ip:
                        return f.get("state")
                if t == "ETH_SRC" and target_mac and c.get("mac") == target_mac:
                    return f.get("state")
        return None

    def mitigation_drop_packets_for(
        self,
        target_ip: Optional[str],
        target_mac: Optional[str],
        flows: Optional[List[dict]] = None,
    ) -> Optional[int]:
        if flows is None:
            flows = self.get_flows()
        total = 0
        matched_any = False
        for f in flows:
            if f.get("priority") != 40000:
                continue
            instructions = f.get("treatment", {}).get("instructions", [])
            is_drop = (not instructions) or all(
                i.get("type") == "NOACTION" for i in instructions
            )
            if not is_drop:
                continue
            crits = f.get("selector", {}).get("criteria", [])
            matched = False
            for c in crits:
                t = c.get("type")
                if t == "IPV4_SRC" and target_ip:
                    if (c.get("ip") or "").split("/")[0] == target_ip:
                        matched = True
                        break
                if t == "ETH_SRC" and target_mac and c.get("mac") == target_mac:
                    matched = True
                    break
            if matched:
                matched_any = True
                try:
                    total += int(f.get("packets", 0) or 0)
                except (TypeError, ValueError):
                    pass
```

```
        return total if matched_any else None

    def push_flow_per_device(self, payload: dict) -> Dict[str, Optional[str]]:
        device_ids = self.get_device_ids()
        out: Dict[str, Optional[str]] = {}
        if not device_ids:
            return out
        futures = {
            self._executor.submit(self.push_flow, did, dict(payload)): did
            for did in device_ids
        }
        for fut in concurrent.futures.as_completed(futures):
            did = futures[fut]
            out[did] = fut.result()
        return out

    def verify_rule_on_devices(
        self, target_ip: Optional[str], target_mac: Optional[str]
    ) -> Dict[str, str]:
        result: Dict[str, str] = {}
        for f in self.get_flows():
            if f.get("priority") != 40000:
                continue
            instr = f.get("treatment", {}).get("instructions", [])
            is_drop = (not instr) or all(i.get("type") == "NOACTION" for i in instr)
            if not is_drop:
                continue
            match = False
            for c in f.get("selector", {}).get("criteria", []):
                t = c.get("type")
                if t == "IPV4_SRC" and target_ip and (c.get("ip") or "").split("/")[0] ==
↪ target_ip:
                    match = True
                    break
                if t == "ETH_SRC" and target_mac and c.get("mac") == target_mac:
                    match = True
                    break
            if match:
                dev = f.get("deviceId")
                if dev:
                    result[dev] = f.get("state", "UNKNOWN")
        return result

    def wait_for_convergence(
        self,
        target_ip: Optional[str],
        target_mac: Optional[str],
        timeout: float = 5.0,
        poll: float = 0.2,
        _clock=time.monotonic,
        _sleep=time.sleep,
    ) -> "SyncReport":
        device_ids = set(self.get_device_ids())
        t0 = _clock()
        seen: Dict[str, float] = {}
        while _clock() - t0 < timeout:
            states = self.verify_rule_on_devices(target_ip, target_mac)
            for dev, st in states.items():
                if st == "ADDED" and dev not in seen:
                    seen[dev] = _clock() - t0
```

```
            if device_ids and set(seen) >= device_ids:
                break
            _sleep(poll)
        return SyncReport(
            devices=list(device_ids),
            added_at=seen,
            converged=bool(device_ids) and set(seen) >= device_ids,
        )


# ============================================================================
# 6. EXTRACAO E ESTATISTICAS
# ============================================================================
def extract_flow_data(f: dict) -> dict:
    pkts = f.get("packets", 0)
    bytes_ = f.get("bytes", 0)
    out: Dict[str, Any] = {
        "mac_src": None,
        "mac_dst": None,
        "ip_src": None,
        "ip_dst": None,
        "ip_proto": None,
        "tcp_dst": None,
        "udp_dst": None,
        "udp_src": None,
        "pkts": pkts,
        "bytes": bytes_,
        "life": f.get("life", 0) or 0,
    }
    for c in f.get("selector", {}).get("criteria", []):
        t = c.get("type")
        if t == "ETH_SRC":
            out["mac_src"] = c.get("mac")
        elif t == "ETH_DST":
            out["mac_dst"] = c.get("mac")
        elif t == "IPV4_SRC":
            ip = c.get("ip")
            out["ip_src"] = ip.split("/")[0] if ip else None
        elif t == "IPV4_DST":
            ip = c.get("ip")
            out["ip_dst"] = ip.split("/")[0] if ip else None
        elif t == "IP_PROTO":
            try:
                out["ip_proto"] = int(c.get("protocol", 0))
            except (TypeError, ValueError):
                pass
        elif t == "TCP_DST":
            try:
                out["tcp_dst"] = int(c.get("tcpPort", 0))
            except (TypeError, ValueError):
                pass
        elif t == "UDP_DST":
            try:
                out["udp_dst"] = int(c.get("udpPort", 0))
            except (TypeError, ValueError):
                pass
        elif t == "UDP_SRC":
            try:
                out["udp_src"] = int(c.get("udpPort", 0))
            except (TypeError, ValueError):
```

```
                pass
    return out


def build_statistics(
    flows: List[dict],
    mac_info: Dict[str, Dict[str, Any]],
    previous: Dict[str, Dict[str, int]],
    interval: float,
    cfg: Optional["Config"] = None,
    gateway_macs: Optional[Set[str]] = None,
    friendly: Optional[Dict[str, str]] = None,
) -> Dict[str, OriginStats]:
    gateway_macs = gateway_macs or set()
    friendly = friendly or {}

    ip_to_mac: Dict[str, str] = {}
    for mac, info in mac_info.items():
        is_gw = mac in gateway_macs
        for ip in info.get("ips", set()):
            if ip in ip_to_mac and is_gw:
                continue
            ip_to_mac[ip] = mac

    gateway_ips: Set[str] = {
        ip for m in gateway_macs for ip in mac_info.get(m, {}).get("ips", set())
    }

    stats: Dict[str, OriginStats] = defaultdict(OriginStats)
    l3_seen: Set[str] = set()

    for f in flows:
        if f.get("priority") == 40000:
            instructions = f.get("treatment", {}).get("instructions", [])
            if not instructions or all(
                i.get("type") == "NOACTION" for i in instructions
            ):
                continue
        c = extract_flow_data(f)
        origin_ip = c["ip_src"]
        origin_mac = c["mac_src"]

        if (origin_ip and origin_mac and origin_mac in gateway_macs
                and origin_ip not in gateway_ips):
            l3_seen.add(origin_ip)
            continue

        if not origin_ip and origin_mac and origin_mac in mac_info:
            ips = mac_info[origin_mac].get("ips", set())
            if ips:
                origin_ip = next(iter(ips))
        if origin_ip and not origin_mac and origin_ip in ip_to_mac:
            origin_mac = ip_to_mac[origin_ip]
        origin_key = origin_ip or origin_mac
        if not origin_key:
            continue
        dst_ip = c["ip_dst"]
        if not dst_ip and c["mac_dst"] and c["mac_dst"] in mac_info:
            ips = mac_info[c["mac_dst"]].get("ips", set())
            if ips:
```

```
                dst_ip = next(iter(ips))
        dst = dst_ip or c["mac_dst"] or "UNKNOWN"
        port = c["tcp_dst"] or c["udp_dst"]
        s = stats[origin_key]
        s.ip_src = origin_ip
        s.mac_src = origin_mac
        s.flows += 1
        s.pkts += c["pkts"]
        s.bytes += c["bytes"]
        _life = c.get("life", 0) or 0
        if _life > 0 and interval > 0:
            _dt = max(float(_life), interval)
            s.life_pkts_rate += c["pkts"] / _dt
            s.life_bytes_rate += c["bytes"] / _dt
        s.destinations.add(dst)
        s.per_flow_pkts.append(c["pkts"])
        if c["ip_proto"]:
            s.ip_protos.add(c["ip_proto"])
        if port:
            s.dst_port_counter[(dst, port)] += 1
        if c["udp_src"]:
            s.udp_src_counter[c["udp_src"]] += 1

    for key, s in stats.items():
        if s.mac_src:
            s.vendor = mac_vendor(s.mac_src)
        real_mac = ip_to_mac.get(s.ip_src) if s.ip_src else None
        if real_mac and real_mac in mac_info:
            info = mac_info[real_mac]
            if info.get("name"):
                s.friendly_name = info.get("name")
            v = info.get("vlan")
            if v and str(v).lower() != "none":
                s.vlan = str(v)
        if not s.friendly_name and s.ip_src:
            s.friendly_name = friendly.get(s.ip_src)
        if not s.friendly_name and s.mac_src in mac_info:
            s.friendly_name = mac_info[s.mac_src].get("name")
        if not s.vlan and s.mac_src in mac_info:
            v = mac_info[s.mac_src].get("vlan")
            if v and str(v).lower() != "none":
                s.vlan = str(v)
        if s.ip_src and s.ip_src in l3_seen:
            s.l3_rewritten = True

    if cfg is not None and cfg.burst_detection_enabled:
        burst_limiar = cfg.flood_min_pkts_rate * cfg.burst_multiplier
    else:
        burst_limiar = float("inf")

    EMA_ALPHA = 0.5
    if interval > 0:
        for key, s in stats.items():
            prev = previous.get(key, {})
            prev_pkts = prev.get("pkts", 0)
            prev_bytes = prev.get("bytes", 0)
            prev_pkts_rate = prev.get("pkts_rate", 0.0)
            prev_bytes_rate = prev.get("bytes_rate", 0.0)
            detector_warming = (len(previous) == 0)
            first_seen = "pkts" not in prev
```

```
1097                if first_seen and detector_warming:
1098                    raw_pkts_rate = 0.0
1099                    raw_bytes_rate = 0.0
1100                elif s.pkts >= prev_pkts:
1101                    raw_pkts_rate = (s.pkts - prev_pkts) / interval
1102                    raw_bytes_rate = (s.bytes - prev_bytes) / interval
1103                else:
1104                    log.debug("Counter reset detectado para %s", key)
1105                    raw_pkts_rate = (s.pkts - prev_pkts) / interval if s.pkts >= prev_pkts
      ↪ else 0.0
1106                    raw_bytes_rate = (s.bytes - prev_bytes) / interval if s.bytes >= prev_
      ↪ bytes else 0.0
1107                burst = raw_pkts_rate >= burst_limiar
1108                s.pkts_rate_raw = raw_pkts_rate
1109                s.bytes_rate_raw = raw_bytes_rate
1110                if burst:
1111                    s.pkts_rate = raw_pkts_rate
1112                    s.bytes_rate = raw_bytes_rate
1113                elif "pkts_rate" in prev:
1114                    s.pkts_rate = EMA_ALPHA * raw_pkts_rate + (1 - EMA_ALPHA) * prev_pkts_rate
1115                    s.bytes_rate = EMA_ALPHA * raw_bytes_rate + (1 - EMA_ALPHA) * prev_bytes_
      ↪ rate
1116                else:
1117                    s.pkts_rate = raw_pkts_rate
1118                    s.bytes_rate = raw_bytes_rate
1119                if raw_pkts_rate > 1000 and cfg is not None:
1120                    caminho = "BURST" if burst else ("EMA" if "pkts_rate" in prev else "FIRST")
      ↪
1121                    log.debug(
1122                        "DIAG rate %s: raw=%.0f prev_pkts=%d cur_pkts=%d delta=%d "
1123                        "interval=%.1f -> pkts_rate=%.0f [%s] (burst_limiar=%.0f flood_thr=%d)
      ↪ ",
1124                        key, raw_pkts_rate, prev_pkts, s.pkts, s.pkts - prev_pkts,
1125                        interval, s.pkts_rate, caminho, burst_limiar, cfg.flood_min_pkts_rate,
1126                    )
1127
1128        if cfg is not None and getattr(cfg, "churn_rescue", True):
1129            _cmin = getattr(cfg, "churn_min_flows", 4)
1130            for key, s in stats.items():
1131                if (s.flows >= _cmin
1132                        and s.life_pkts_rate >= cfg.flood_min_pkts_rate
1133                        and s.pkts_rate_raw < cfg.flood_min_pkts_rate):
1134                    s.pkts_rate_raw = max(s.pkts_rate_raw, s.life_pkts_rate)
1135                    s.bytes_rate_raw = max(s.bytes_rate_raw, s.life_bytes_rate)
1136
1137        return dict(stats)
1138
1139
1140    def build_destination_statistics(
1141        flows: List[dict],
1142        mac_info: Dict[str, Dict[str, Any]],
1143        previous_dst: Dict[str, Dict[str, int]],
1144        interval: float,
1145        cfg: "Config",
1146        gateway_macs: Set[str],
1147        friendly: Optional[Dict[str, str]] = None,
1148        gateway_faces: Optional[Dict[str, Dict[str, Dict[str, Optional[str]]]]] = None,
1149    ) -> Dict[str, DestinationStats]:
1150        """v18/v19beta - Conta por destino REAL, direto dos flows."""
1151        friendly = friendly or {}
```

```
    gateway_faces = gateway_faces or {}
    EMA_ALPHA = 0.5

    ip_to_mac: Dict[str, str] = {
        ip: mac for mac, info in mac_info.items() for ip in info.get("ips", set())
    }

    cur_pkts: Dict[str, int] = defaultdict(int)
    cur_bytes: Dict[str, int] = defaultdict(int)
    origins: Dict[str, Set[str]] = defaultdict(set)
    is_transit: Dict[str, bool] = {}
    gw_mac_of: Dict[str, str] = {}
    transit_src_vlan: Dict[str, Counter] = defaultdict(Counter)
    dst_vlan: Dict[str, Optional[str]] = {}
    transit_origins_by_vlan: Dict[str, Set[str]] = defaultdict(set)
    real_dst_macs: Set[str] = set()

    def _vlan_of_mac(m: Optional[str]) -> Optional[str]:
        if m and m in mac_info:
            v = mac_info[m].get("vlan")
            if v and str(v).lower() != "none":
                return str(v)
        return None

    for f in flows:
        if f.get("priority") == 40000:
            instr = f.get("treatment", {}).get("instructions", [])
            if not instr or all(i.get("type") == "NOACTION" for i in instr):
                continue
        c = extract_flow_data(f)
        origin_key = c["ip_src"] or c["mac_src"]
        if not origin_key:
            continue

        mac_dst = c["mac_dst"]
        ip_dst = c["ip_dst"]

        try:
            pkts = int(f.get("packets", 0) or 0)
            bytes_ = int(f.get("bytes", 0) or 0)
        except (TypeError, ValueError):
            pkts, bytes_ = 0, 0

        if mac_dst and mac_dst in gateway_macs:
            dst_label = f"gw:{mac_dst}"
            is_transit[dst_label] = True
            gw_mac_of[dst_label] = mac_dst
            src_vlan = _vlan_of_mac(c["mac_src"])
            if src_vlan:
                transit_src_vlan[dst_label][src_vlan] += max(pkts, 1)
                transit_origins_by_vlan[src_vlan].add(origin_key)
        else:
            if not ip_dst and mac_dst and mac_dst in mac_info:
                ips = mac_info[mac_dst].get("ips", set())
                if len(ips) == 1:
                    ip_dst = next(iter(ips))
                elif len(ips) > 1:
                    ip_dst = sorted(ips)[0]
            dst_label = ip_dst or mac_dst or "UNKNOWN"
            is_transit.setdefault(dst_label, False)
```

```
            if mac_dst:
                real_dst_macs.add(mac_dst)
            if dst_label not in dst_vlan:
                m = mac_dst or ip_to_mac.get(ip_dst or "")
                dst_vlan[dst_label] = _vlan_of_mac(m)

        if dst_label == "UNKNOWN":
            continue

        cur_pkts[dst_label] += pkts
        cur_bytes[dst_label] += bytes_
        origins[dst_label].add(origin_key)

    all_transit_origins: Set[str] = set()
    for _vl, _orgs in transit_origins_by_vlan.items():
        all_transit_origins |= _orgs

    gateway_identities: Set[str] = set(gateway_macs)
    for _gwm in gateway_macs:
        for _ip in mac_info.get(_gwm, {}).get("ips", set()):
            gateway_identities.add(_ip)
    all_transit_origins -= gateway_identities

    origin_vlan: Dict[str, Optional[str]] = {}
    for _vl, _orgs in transit_origins_by_vlan.items():
        for _o in _orgs:
            origin_vlan[_o] = _vl

    out: Dict[str, DestinationStats] = {}
    for dst_label, pkts in cur_pkts.items():
        d = DestinationStats(dst=dst_label)
        d.total_pkts = pkts
        d.origins = origins[dst_label]
        d.is_transit = is_transit.get(dst_label, False)

        if d.is_transit:
            gw_mac = gw_mac_of.get(dst_label, dst_label.replace("gw:", ""))
            vlan = None
            if transit_src_vlan[dst_label]:
                vlan = transit_src_vlan[dst_label].most_common(1)[0][0]
            face = (gateway_faces.get(gw_mac, {}) or {}).get(str(vlan)) if vlan else None
            if not face:
                faces = gateway_faces.get(gw_mac, {})
                if faces:
                    vlan = sorted(faces.keys())[0]
                    face = faces[vlan]
            if face and face.get("name"):
                d.friendly_name = face["name"]
                d.display_ip = face.get("ip")
            else:
                d.friendly_name = "gateway"
                d.display_ip = gw_mac
            d.vlan = (str(vlan) if vlan else None)
            d.effective_origins = set(d.origins)
        else:
            d.friendly_name = friendly.get(dst_label)
            d.display_ip = dst_label
            d.vlan = dst_vlan.get(dst_label)
            d.effective_origins = {o for o in d.origins if o not in gateway_identities}

```

```
        prev = previous_dst.get(dst_label, {})
        prev_pkts = prev.get("pkts", 0)
        prev_bytes = prev.get("bytes", 0)
        prev_rate = prev.get("pkts_rate", 0.0)
        prev_brate = prev.get("bytes_rate", 0.0)

        if interval > 0:
            detector_warming = (len(previous_dst) == 0)
            first_seen = "pkts" not in prev
            if first_seen and detector_warming:
                raw_rate = 0.0
                raw_brate = 0.0
            elif pkts >= prev_pkts:
                raw_rate = (pkts - prev_pkts) / interval
                raw_brate = (cur_bytes[dst_label] - prev_bytes) / interval
            else:
                raw_rate = 0.0
                raw_brate = 0.0
            if "pkts_rate" in prev:
                d.total_pkts_rate = EMA_ALPHA * raw_rate + (1 - EMA_ALPHA) * prev_rate
                d.total_bytes_rate = EMA_ALPHA * raw_brate + (1 - EMA_ALPHA) * prev_brate
            else:
                d.total_pkts_rate = raw_rate
                d.total_bytes_rate = raw_brate

        _recover_thr = (cfg.dest_distributed_min_total_pps
                        if getattr(cfg, "residual_hysteresis", True)
                        else cfg.flood_min_pkts_rate)
        if not d.is_transit and d.total_pkts_rate >= _recover_thr:
            dvlan = dst_vlan.get(dst_label)
            recovered = {
                o for o in all_transit_origins
                if o != dst_label and o != d.display_ip
                and (dvlan is None or origin_vlan.get(o) != dvlan)
            }
            d.effective_origins = d.effective_origins | recovered

        out[dst_label] = d

    previous_dst.clear()
    for dst_label, d in out.items():
        previous_dst[dst_label] = {
            "pkts": d.total_pkts,
            "bytes": cur_bytes[dst_label],
            "pkts_rate": d.total_pkts_rate,
            "bytes_rate": d.total_bytes_rate,
        }
    return out


def compute_target_impact(dst_stats: Dict[str, DestinationStats]) -> "TargetImpact":
    """v19beta - separa o ENTREGUE (destinos reais) do TRANSITO (gateway)."""
    ti = TargetImpact()
    delivered_rate_total = 0.0
    for label, d in dst_stats.items():
        if getattr(d, "is_transit", False):
            ti.transit_pkts += d.total_pkts
            ti.transit_rate += d.total_pkts_rate
        else:
            ti.per_target[label] = {
```

```
                "pkts": float(d.total_pkts),
                "rate": d.total_pkts_rate,
            }
            delivered_rate_total += d.total_pkts_rate
    denom = delivered_rate_total + ti.transit_rate
    if denom > 0:
        ti.efficacy_estimate = 1.0 - (delivered_rate_total / denom)
    return ti


# ============================================================================
# 7. CLASSIFICADORES (puros, faceis de testar)
# ============================================================================
def classify_scan(s: OriginStats, cfg: Config) -> bool:
    if cfg.skip_icmp_scan and s.ip_protos == {1}:
        return False
    return (
        s.flows >= cfg.scan_min_flows
        and len(s.destinations) >= cfg.scan_min_destinations
        and bool(s.per_flow_pkts)
        and statistics.mean(s.per_flow_pkts) <= cfg.scan_max_pkts_avg
    )


def is_transit_node(
    origin_key: str,
    stats: Dict[str, "OriginStats"],
    dst_stats: Dict[str, "DestinationStats"],
    cfg: Config,
) -> bool:
    if not cfg.transit_detection_enabled:
        return False
    src_rate = stats[origin_key].pkts_rate if origin_key in stats else 0
    dst_rate = dst_stats[origin_key].total_pkts_rate if origin_key in dst_stats else 0
    return src_rate >= cfg.transit_threshold_pps and dst_rate >= cfg.transit_threshold_pps


def classify_brute(s: OriginStats, cfg: Config) -> Optional[int]:
    for (_dst, port), count in s.dst_port_counter.items():
        if port in cfg.brute_ports and count >= cfg.brute_min_conns:
            return port
    return None


def _effective_pkts_rate(s: OriginStats) -> float:
    return max(s.pkts_rate, s.pkts_rate_raw)


def _effective_bytes_rate(s: OriginStats) -> float:
    return max(s.bytes_rate, s.bytes_rate_raw)


def classify_flood(s: OriginStats, cfg: Config) -> bool:
    return (
        (_effective_pkts_rate(s) >= cfg.flood_min_pkts_rate
         or _effective_bytes_rate(s) >= cfg.flood_min_bytes_rate)
        and len(s.destinations) <= cfg.flood_max_destinations
    )


```

```
def classify_amplification(s: OriginStats, cfg: Config) -> bool:
    if s.pkts == 0:
        return False
    avg_size = s.bytes / s.pkts
    has_amp_port = any(p in cfg.amp_ports for p in s.udp_src_counter.keys())
    return (
        has_amp_port
        and avg_size >= cfg.amp_min_avg_packet_size
        and _effective_pkts_rate(s) >= cfg.amp_min_pkts_rate
    )


def classify_dns_water_torture(s: OriginStats, cfg: Config) -> bool:
    dns_queries = sum(
        count for (_dst, port), count in s.dst_port_counter.items() if port == 53
    )
    return (
        dns_queries >= cfg.dns_torture_min_queries
        and _effective_pkts_rate(s) >= cfg.dns_torture_min_rate
    )


def classify_gre_flood(s: OriginStats, cfg: Config) -> bool:
    return 47 in s.ip_protos and _effective_pkts_rate(s) >= cfg.gre_min_pkts_rate


def classify_distributed_attack(d: DestinationStats, cfg: Config) -> bool:
    """v18/v19beta - usa as origens EFETIVAS (correlacao pre/pos-L3)."""
    if getattr(d, "is_transit", False):
        return False
    n_orig = len(getattr(d, "effective_origins", None) or d.origins)
    return (
        n_orig >= cfg.dest_distributed_min_origins
        and d.total_pkts_rate >= cfg.dest_distributed_min_total_pps
    )


def target_sort_rank(d: DestinationStats, cfg: Config) -> int:
    """v19beta - prioridade de ordenacao do painel 'Alvos Sob Ataque'."""
    if getattr(d, "is_transit", False):
        return 3
    if classify_distributed_attack(d, cfg):
        return 0
    if d.total_pkts_rate >= cfg.flood_min_pkts_rate:
        return 1
    return 2


def classify_scan_subtype(s: OriginStats, cfg: Config) -> Optional[str]:
    if not classify_scan(s, cfg):
        return None
    if s.ip_protos == {1}:
        return "PING_SWEEP"
    if 6 in s.ip_protos or 17 in s.ip_protos:
        return "PORT_SCAN"
    return "SCAN"


def get_status(s: OriginStats, cfg: Config) -> str:
    if classify_gre_flood(s, cfg):
```

```
1452            return "GRE-FLOOD"
1453        if classify_dns_water_torture(s, cfg):
1454            return "DNS-TORTURE"
1455        if classify_flood(s, cfg):
1456            return "FLOOD"
1457        scan_sub = classify_scan_subtype(s, cfg)
1458        if scan_sub:
1459            return scan_sub
1460        if classify_amplification(s, cfg):
1461            return "UDP-AMP"
1462        if classify_brute(s, cfg):
1463            return "BRUTE"
1464        return "NORMAL"
1465
1466
1467    # ============================================================================
1468    # 8. MITIGACAO
1469    # ============================================================================
1470    class MitigationManager:
1471        def __init__(self, cfg: Config, onos: OnosClient, prom: Optional["PrometheusExporter"]
        ↪  = None):
1472            self.cfg = cfg
1473            self.onos = onos
1474            self.prom = prom
1475            self.active_blocks: Dict[str, BlockState] = {}
1476            self._dist_victims: Dict[str, int] = {}  # v19.3: alvo -> ciclos restantes de
        ↪ histerese
1477            self.recent_actions: Deque[str] = deque(maxlen=20)
1478            self._mitigation_window: Deque[float] = deque(maxlen=100)
1479            self.last_target_impact: Optional[TargetImpact] = None  # v19beta
1480            self._load_state()
1481
1482        def _load_state(self) -> None:
1483            p = Path(self.cfg.state_filename)
1484            if not p.exists():
1485                return
1486            if self.cfg.mode != "ips":
1487                log.info("Modo IDS: estado de bloqueios do disco ignorado (IDS nao bloqueia)")
1488                return
1489            try:
1490                data = json.loads(p.read_text())
1491                for k, v in data.get("active_blocks", {}).items():
1492                    known = {f for f in BlockState.__dataclass_fields__}
1493                    self.active_blocks[k] = BlockState(**{kk: vv for kk, vv in v.items() if kk
        ↪  in known})
1494                log.info("Estado restaurado: %d bloqueios ativos", len(self.active_blocks))
1495            except Exception:
1496                log.exception("Falha ao carregar estado, iniciando vazio")
1497
1498        def _save_state(self) -> None:
1499            try:
1500                data = {
1501                    "active_blocks": {
1502                        k: v.__dict__ for k, v in self.active_blocks.items()
1503                    }
1504                }
1505                Path(self.cfg.state_filename).write_text(json.dumps(data, default=list))
1506            except Exception:
1507                log.exception("Falha ao salvar estado")
1508
```

```
    def is_allowlisted(self, ip: Optional[str], mac: Optional[str]) -> bool:
        if ip and ip in self.cfg.allowlist_ips:
            return True
        if mac and mac in self.cfg.allowlist_macs:
            return True
        return False

    def _within_rate_limit(self) -> bool:
        now = time.time()
        while self._mitigation_window and now - self._mitigation_window[0] > 60:
            self._mitigation_window.popleft()
        return len(self._mitigation_window) < self.cfg.max_mitigations_per_minute

    def _record_mitigation(self) -> None:
        self._mitigation_window.append(time.time())

    def _should_block_by_mac(self, attack_types: List[str], s: OriginStats) -> bool:
        if not s.mac_src:
            return False
        return self.cfg.block_strategy == "mac"

    def _drop(
        self,
        target_ip: Optional[str],
        target_mac: Optional[str],
        is_brute: bool = False,
        brute_port: Optional[int] = None,
    ) -> List[str]:
        target_id = target_ip or target_mac
        if not target_id:
            return []
        flow_id_string = f"mitigation_{target_id.replace(':', '_').replace('.', '_')}"
        is_ipv4 = "." in target_id and ":" not in target_id
        criteria: List[dict] = []
        if is_ipv4:
            criteria.append({"type": "ETH_TYPE", "ethType": "0x0800"})
            criteria.append({"type": "IPV4_SRC", "ip": f"{target_id}/32"})
        else:
            criteria.append({"type": "ETH_SRC", "mac": target_id})
        if is_brute and brute_port:
            flow_id_string += f"_port_{brute_port}"
            criteria.extend(
                [
                    {"type": "IP_PROTO", "protocol": 6},
                    {"type": "TCP_DST", "tcpPort": brute_port},
                ]
            )
        payload = {
            "priority": 40000,
            "timeout": 0,
            "isPermanent": True,
            "appId": "org.onosproject.cli",
            "treatment": {"instructions": []},
            "selector": {"criteria": criteria},
            "id": flow_id_string,
        }
        return self.onos.push_flow_to_all(payload)

    def _undrop(self, state: BlockState) -> int:
        return self.onos.delete_mitigation_flows_for(
```

```
            target_ip=state.target_ip, target_mac=state.target_mac
        )

    def reconcile_orphans(self) -> int:
        tracked_ips = {b.target_ip for b in self.active_blocks.values() if b.target_ip}
        tracked_macs = {b.target_mac for b in self.active_blocks.values() if b.target_mac}
        removed = 0
        for f in self.onos.get_flows():
            if f.get("priority") != 40000:
                continue
            instr = f.get("treatment", {}).get("instructions", [])
            is_drop = (not instr) or all(i.get("type") == "NOACTION" for i in instr)
            if not is_drop:
                continue  # protege LLDP/BDDP/ARP (treatment OUTPUT)
            ip = mac = None
            for c in f.get("selector", {}).get("criteria", []):
                t = c.get("type")
                if t == "IPV4_SRC":
                    ip = (c.get("ip") or "").split("/")[0]
                elif t == "ETH_SRC":
                    mac = c.get("mac")
            if not ip and not mac:
                continue  # nao e' DROP ancorado em host
            is_orphan = (ip not in tracked_ips) and (mac not in tracked_macs)
            if is_orphan:
                dev = f.get("deviceId")
                fid = f.get("id")
                if dev and fid and self.onos.delete_flow(dev, fid):
                    removed += 1
                    log.info("Orfa removida: %s/%s (ip=%s mac=%s)", dev, fid, ip, mac)
        if removed:
            log.warning("Reconciliacao de boot: %d regra(s) orfa(s) removida(s)", removed)
        return removed

    def _escalate_to_mac(self, state: BlockState, origin_key: str,
                         action_logs: List[str]) -> bool:
        mac = state.target_mac
        if not mac:
            return False
        new_ids = self._drop(None, mac)
        if not new_ids:
            return False
        old_ip = state.target_ip
        if old_ip:
            try:
                self.onos.delete_mitigation_flows_for(target_ip=old_ip, target_mac=None)
            except Exception:
                log.debug("Falha ao remover regra IP na escalada")
        state.flow_ids = new_ids
        state.anchor = "mac"
        state.escalated = True
        state.spoofed = True
        state.target_ip = None
        state.last_drop_pkts = None
        state.zero_rate_count = 0
        if "/SPOOFED" not in state.attack_label:
            state.attack_label = f"{state.attack_label}/SPOOFED"
        hh = datetime.datetime.now().strftime("%H:%M:%S")
        rotulo = fmt_label(state.friendly_name, origin_key, self.cfg.show_friendly)
        msg = (f"[{hh}] [ESCALADA] {state.attack_label} de {rotulo}: regra por "
```

```
                f"IP ineficaz (origem forjada) -> bloqueio por ETH_SRC {mac}")
        action_logs.append(msg)
        log.info(msg)
        return True

    def _engage_origin(self, origin_key: str, s: OriginStats,
                       attack_label: str, action_logs: List[str]) -> bool:
        """v19fix - aplica BLOQUEIO (IPS) ou ALERTA (IDS) para UMA origem."""
        if origin_key in self.active_blocks:
            return False
        ip_label = fmt_label(s.friendly_name, s.ip_src or s.mac_src, self.cfg.show_
    ↪ friendly)
        hhmmss = datetime.datetime.now().strftime("%H:%M:%S")

        if self.is_allowlisted(s.ip_src, s.mac_src):
            msg = f"[{hhmmss}] [SKIP-ALLOWLIST] {attack_label} de {ip_label}"
            action_logs.append(msg)
            log.info(msg)
            return False

        if self.cfg.mode == "ips" and not self._within_rate_limit():
            msg = f"[{hhmmss}] [RATE-LIMIT] {attack_label} de {ip_label} adiado"
            action_logs.append(msg)
            log.warning(msg)
            return False

        if self.cfg.mode == "ips":
            epoch_detect = time.time()
            block_by_mac = self._should_block_by_mac([attack_label], s)
            if block_by_mac:
                drop_ip, drop_mac, anchor = None, s.mac_src, f"ETH_SRC {s.mac_src}"
            else:
                drop_ip, drop_mac, anchor = s.ip_src, s.mac_src, f"IPV4_SRC {s.ip_src}"
            t0 = time.perf_counter()
            flow_ids = self._drop(drop_ip, drop_mac)
            latency_ms = (time.perf_counter() - t0) * 1000
            epoch_action = time.time()
            if not flow_ids:
                msg = f"[{hhmmss}] [FALHA] {attack_label} de {ip_label} (push falhou)"
                action_logs.append(msg)
                log.error(msg)
                return False
            self.active_blocks[origin_key] = BlockState(
                flow_ids=flow_ids,
                zero_rate_count=0,
                attack_label=attack_label,
                blocked_at=epoch_action,
                target_ip=None if block_by_mac else s.ip_src,
                target_mac=s.mac_src,
                last_drop_pkts=None,
                anchor=("mac" if block_by_mac else "ip"),
                friendly_name=s.friendly_name,
            )
            self._record_mitigation()
            if self.cfg.sync_verify:
                try:
                    states = self.onos.verify_rule_on_devices(
                        None if block_by_mac else s.ip_src, s.mac_src
                    )
                    if states:
```

```
                        resumo = ", ".join(f"{dv[-4:]}:{st}" for dv, st in states.items())
                        log.info("[SYNC] regra de %s em %d switch(es): %s",
                                 ip_label, len(states), resumo)
                except Exception:
                    log.debug("Falha na verificacao de sincronizacao")
            msg = (f"[{hhmmss}] [BLOQUEIO] {attack_label} de {ip_label} "
                   f"via {anchor} (REST {latency_ms:.2f}ms)")
            action_logs.append(msg)
            log.warning(msg)
            self._log_csv(
                s.ip_src, s.mac_src, attack_label, "BLOQUEIO",
                epoch_detect=epoch_detect, epoch_action=epoch_action,
                latency_rest_ms=latency_ms,
            )
            if self.prom:
                self.prom.set_latency(ip_label, latency_ms)
            return True

        # modo IDS: alerta, sem regra
        epoch_detect = time.time()
        msg = f"[{hhmmss}] [ALERTA-IDS] {attack_label} de {ip_label}"
        action_logs.append(msg)
        log.info(msg)
        self.active_blocks[origin_key] = BlockState(
            flow_ids=[],
            zero_rate_count=0,
            attack_label=attack_label,
            blocked_at=epoch_detect,
            target_ip=s.ip_src,
            target_mac=s.mac_src,
            last_drop_pkts=None,
            friendly_name=s.friendly_name,
        )
        self._log_csv(s.ip_src, s.mac_src, attack_label, "ALERTA",
                      epoch_detect=epoch_detect)
        return True

    def step(self, stats: Dict[str, OriginStats], dst_stats_param: Optional[Dict[str, "
    ↪ DestinationStats"]] = None,
             flows: Optional[List[dict]] = None) -> List[str]:
        if dst_stats_param is None:
            dst_stats_param = {}
        action_logs: List[str] = []
        volumetric_block = False
        router_macs = set()
        if self.cfg.auto_detect_routers:
            try:
                router_macs = self.onos.get_router_macs(
                    ttl=self.cfg.topology_cache_ttl
                )
            except Exception:
                log.debug("Nao foi possivel obter router MACs")
        for origin_key, s in stats.items():
            attack_types: List[str] = []
            brute_port = classify_brute(s, self.cfg)
            if classify_flood(s, self.cfg):
                attack_types.append("FLOOD")
            if classify_gre_flood(s, self.cfg):
                attack_types.append("GRE-FLOOD")
            if classify_dns_water_torture(s, self.cfg):
```

```
1747                 attack_types.append("DNS-TORTURE")
1748             if classify_scan(s, self.cfg):
1749                 attack_types.append("SCAN")
1750             if classify_amplification(s, self.cfg):
1751                 attack_types.append("UDP-AMP")
1752             if brute_port:
1753                 attack_types.append(f"BRUTE_PORT_{brute_port}")
1754 
1755             volume_attack = any(t in ("FLOOD", "GRE-FLOOD", "UDP-AMP") for t in attack_
    ↪ types)
1756             if volume_attack:
1757                 volumetric_block = True
1758             if not volume_attack:
1759                 if is_transit_node(origin_key, stats, dst_stats_param, self.cfg):
1760                     continue
1761                 if self.cfg.auto_detect_routers and s.mac_src in router_macs:
1762                     continue
1763 
1764             if not attack_types or origin_key in self.active_blocks:
1765                 continue
1766 
1767             attack_label = " + ".join(attack_types)
1768             ip_label = fmt_label(s.friendly_name, s.ip_src or s.mac_src, self.cfg.show_
    ↪ friendly)
1769             hhmmss = datetime.datetime.now().strftime("%H:%M:%S")
1770 
1771             if self.is_allowlisted(s.ip_src, s.mac_src):
1772                 msg = f"[{hhmmss}] [SKIP-ALLOWLIST] {attack_label} de {ip_label}"
1773                 action_logs.append(msg)
1774                 log.info(msg)
1775                 continue
1776 
1777             if self.cfg.mode == "ips" and not self._within_rate_limit():
1778                 msg = f"[{hhmmss}] [RATE-LIMIT] {attack_label} de {ip_label} adiado"
1779                 action_logs.append(msg)
1780                 log.warning(msg)
1781                 continue
1782 
1783             if self.cfg.mode == "ips":
1784                 epoch_detect = time.time()
1785                 block_by_mac = self._should_block_by_mac(attack_types, s)
1786                 if block_by_mac:
1787                     drop_ip, drop_mac, anchor = None, s.mac_src, f"ETH_SRC {s.mac_src}"
1788                 else:
1789                     drop_ip, drop_mac, anchor = s.ip_src, s.mac_src, f"IPV4_SRC {s.ip_src}
    ↪ "
1790                 t0 = time.perf_counter()
1791                 flow_ids = self._drop(
1792                     drop_ip,
1793                     drop_mac,
1794                     is_brute=(len(attack_types) == 1 and attack_types[0].startswith("BRUTE
    ↪ ")),
1795                     brute_port=brute_port,
1796                 )
1797                 latency_ms = (time.perf_counter() - t0) * 1000
1798                 epoch_action = time.time()
1799                 if flow_ids:
1800                     self.active_blocks[origin_key] = BlockState(
1801                         flow_ids=flow_ids,
1802                         zero_rate_count=0,
```

```
                        attack_label=attack_label,
                        blocked_at=epoch_action,
                        target_ip=None if block_by_mac else s.ip_src,
                        target_mac=s.mac_src,
                        last_drop_pkts=None,
                        anchor=("mac" if block_by_mac else "ip"),
                        friendly_name=s.friendly_name,  # v18
                    )
                    self._record_mitigation()
                    if self.cfg.sync_verify:
                        try:
                            states = self.onos.verify_rule_on_devices(
                                None if block_by_mac else s.ip_src, s.mac_src
                            )
                            if states:
                                resumo = ", ".join(f"{d[-4:]}:{st}" for d, st in states.
    ↪ items())
                                log.info("[SYNC] regra de %s em %d switch(es): %s",
                                         ip_label, len(states), resumo)
                        except Exception:
                            log.debug("Falha na verificacao de sincronizacao")
                    msg = (
                        f"[{hhmmss}] [BLOQUEIO] {attack_label} de {ip_label} "
                        f"via {anchor} (REST {latency_ms:.2f}ms)"
                    )
                    action_logs.append(msg)
                    log.warning(msg)
                    self._log_csv(
                        s.ip_src, s.mac_src, attack_label, "BLOQUEIO",
                        epoch_detect=epoch_detect, epoch_action=epoch_action,
                        latency_rest_ms=latency_ms,
                    )
                    if self.prom:
                        self.prom.set_latency(ip_label, latency_ms)
                else:
                    msg = f"[{hhmmss}] [FALHA] {attack_label} de {ip_label} (push falhou)"
                    action_logs.append(msg)
                    log.error(msg)
            else:
                epoch_detect = time.time()
                msg = f"[{hhmmss}] [ALERTA-IDS] {attack_label} de {ip_label}"
                action_logs.append(msg)
                log.info(msg)
                self.active_blocks[origin_key] = BlockState(
                    flow_ids=[],
                    zero_rate_count=0,
                    attack_label=attack_label,
                    blocked_at=epoch_detect,
                    target_ip=s.ip_src,
                    target_mac=s.mac_src,
                    last_drop_pkts=None,
                    friendly_name=s.friendly_name,  # v18
                )
                self._log_csv(
                    s.ip_src, s.mac_src, attack_label, "ALERTA",
                    epoch_detect=epoch_detect,
                )

        if getattr(self.cfg, "block_distributed", True):
            if getattr(self.cfg, "residual_hysteresis", True):
```

```
                for _t in list(self._dist_victims):
                    self._dist_victims[_t] -= 1
                    if self._dist_victims[_t] <= 0:
                        del self._dist_victims[_t]
                if volumetric_block:
                    for _dl, _d in dst_stats_param.items():
                        if getattr(_d, "is_transit", False):
                            continue
                        if _d.total_pkts_rate >= self.cfg.dest_distributed_min_total_pps:
                            self._dist_victims[_dl] = self.cfg.distributed_victim_ttl
            for dst_label, d in dst_stats_param.items():
                if getattr(d, "is_transit", False):
                    continue
                eff = getattr(d, "effective_origins", None) or d.origins
                if getattr(self.cfg, "residual_hysteresis", True):
                    _rate_ok = d.total_pkts_rate >= self.cfg.dest_distributed_min_total_
    ↪ pps
                    if _rate_ok and len(eff) >= 2:
                        self._dist_victims[dst_label] = self.cfg.distributed_victim_ttl
                    _hot = dst_label in self._dist_victims
                    if _hot and _rate_ok:
                        self._dist_victims[dst_label] = self.cfg.distributed_victim_ttl
                    aggregate_flood = _rate_ok and (len(eff) >= 2 or _hot)
                else:
                    aggregate_flood = (
                        d.total_pkts_rate >= self.cfg.flood_min_pkts_rate
                        and len(eff) >= 2
                    )
                if not (aggregate_flood or classify_distributed_attack(d, self.cfg)):
                    continue
                for origin_key in eff:
                    if origin_key in self.active_blocks:
                        continue
                    s = stats.get(origin_key)
                    if s is None:
                        continue
                    if self.cfg.auto_detect_routers and s.mac_src in router_macs:
                        continue
                    self._engage_origin(origin_key, s, "FLOOD/DIST", action_logs)

        use_counter = self.cfg.unblock_by_drop_counter and self.cfg.mode == "ips"
        to_remove: List[str] = []
        for origin_key, state in list(self.active_blocks.items()):
            attacker_active: bool

            if use_counter:
                drop_pkts = self.onos.mitigation_drop_packets_for(
                    state.target_ip, state.target_mac, flows=flows
                )
                if drop_pkts is None:
                    attacker_active = False
                else:
                    if state.last_drop_pkts is None:
                        attacker_active = True
                        state.drop_pps = 0.0
                    else:
                        delta = drop_pkts - state.last_drop_pkts
                        attacker_active = delta > self.cfg.unblock_residual_threshold
                        state.drop_pps = max(0.0, delta) / self.cfg.interval if self.cfg.
    ↪ interval > 0 else 0.0
```

```
                    state.last_drop_pkts = drop_pkts
                    state.drop_pkts_total = drop_pkts
                    state.last_active = attacker_active
                    EMA_A = 0.4
                    state.drop_pps_ema = (
                        EMA_A * state.drop_pps + (1 - EMA_A) * state.drop_pps_ema
                    )
            else:
                current = stats.get(origin_key)
                attacker_active = not (
                    current is None or current.pkts_rate <= self.cfg.zero_rate_threshold
                )
                state.observed_pps = current.pkts_rate if current else 0.0

            if (use_counter and self.cfg.block_strategy == "auto"
                    and state.anchor == "ip" and not state.escalated
                    and state.last_drop_pkts is not None
                    and any(v in state.attack_label for v in self.cfg.volumetric_labels)):
                cur = stats.get(origin_key)
                attack_flowing = (
                    cur is not None and cur.pkts_rate >= self.cfg.flood_min_pkts_rate
                )
                if (not attacker_active) and attack_flowing:
                    state.spoof_evidence += 1
                    state.zero_rate_count = 0
                    if state.spoof_evidence >= self.cfg.escalation_cycles:
                        self._escalate_to_mac(state, origin_key, action_logs)
                    continue
                else:
                    state.spoof_evidence = 0

            if attacker_active:
                state.zero_rate_count = 0
                continue

            state.zero_rate_count += 1
            clear_cycles = (
                self.cfg.mitigation_zero_rate_cycles if self.cfg.mode == "ips"
                else self.cfg.ids_alert_clear_cycles
            )
            if state.zero_rate_count >= clear_cycles:
                n_deleted = 0
                if self.cfg.mode == "ips":
                    n_deleted = self._undrop(state)
                hhmmss_now = datetime.datetime.now().strftime("%H:%M:%S")
                suffix = f" ({n_deleted} regra(s) removida(s))" if n_deleted else ""
                rotulo = fmt_label(state.friendly_name, origin_key, self.cfg.show_friendly)
    ↪
                msg = (
                    f"[{hhmmss_now}] [UNBLOCK] {state.attack_label} de "
                    f"{rotulo} normalizou{suffix}"
                )
                action_logs.append(msg)
                log.info(msg)
                self._log_csv(
                    state.target_ip, state.target_mac, state.attack_label,
                    "UNBLOCK", epoch_detect=time.time(),
                )
                to_remove.append(origin_key)
        for k in to_remove:
```

```
            del self.active_blocks[k]

        self.recent_actions.extend(action_logs)
        self._save_state()
        return action_logs

    def _log_csv(
        self,
        ip: Optional[str],
        mac: Optional[str],
        attack: str,
        action: str,
        epoch_detect: Optional[float] = None,
        epoch_action: Optional[float] = None,
        latency_rest_ms: Optional[float] = None,
    ) -> None:
        if epoch_detect is None:
            epoch_detect = time.time()
        ts_humano = datetime.datetime.fromtimestamp(epoch_detect).strftime(
            "%Y-%m-%d %H:%M:%S.%f"
        )[:-3]
        path = Path(self.cfg.csv_filename)
        new = not path.exists()
        with open(path, "a", newline="") as f:
            w = csv.writer(f)
            if new:
                w.writerow(
                    ["epoch_detect", "epoch_action", "timestamp_humano",
                     "ip", "mac", "tipo_ataque", "acao", "latencia_rest_ms"]
                )
            w.writerow(
                [
                    f"{epoch_detect:.3f}",
                    f"{epoch_action:.3f}" if epoch_action is not None else "N/A",
                    ts_humano,
                    ip or "N/A",
                    mac or "N/A",
                    attack,
                    action,
                    f"{latency_rest_ms:.2f}" if latency_rest_ms is not None else "N/A",
                ]
            )


# ======================================================================
# 9. PROMETHEUS
# ======================================================================
class PrometheusExporter:
    def __init__(self, port: int):
        if not PROMETHEUS_ENABLED:
            self.enabled = False
            return
        self.enabled = True
        self.g_pkts_rate = Gauge("sdn_pkts_rate", "Taxa de pacotes/s", ["ip_src", "status"
    ↪ ])
        self.g_bytes_rate = Gauge("sdn_bytes_rate", "Taxa de bytes/s", ["ip_src", "status"
    ↪ ])
        self.g_flows = Gauge("sdn_active_flows", "Numero de flows", ["ip_src", "status"])
        self.g_latency = Gauge("sdn_mitigation_latency_ms", "Latencia (ms)", ["ip_src"])
        start_http_server(port)
```

```
2037          log.info("Prometheus exporter iniciado na porta %d", port)
2038
2039      def update(self, stats: Dict[str, OriginStats], cfg: Config) -> None:
2040          if not self.enabled:
2041              return
2042          for _, s in stats.items():
2043              ip_label = s.ip_src or s.mac_src or "Desconhecido"
2044              status = get_status(s, cfg)
2045              self.g_pkts_rate.labels(ip_src=ip_label, status=status).set(s.pkts_rate)
2046              self.g_bytes_rate.labels(ip_src=ip_label, status=status).set(s.bytes_rate)
2047              self.g_flows.labels(ip_src=ip_label, status=status).set(s.flows)
2048
2049      def set_latency(self, ip: str, ms: float) -> None:
2050          if self.enabled:
2051              self.g_latency.labels(ip_src=ip).set(ms)
2052
2053
2054  # ======================================================================
2055  # 10. DASHBOARD (rich)
2056  # ======================================================================
2057  STATUS_COLOR = {
2058      "PING_SWEEP": "yellow",
2059      "PORT_SCAN": "bold yellow",
2060      "FLOOD": "bold red",
2061      "GRE-FLOOD": "bold red",
2062      "DNS-TORTURE": "bold red",
2063      "SCAN": "yellow",
2064      "UDP-AMP": "magenta",
2065      "BRUTE": "bold magenta",
2066      "NORMAL": "green",
2067  }
2068
2069  STATUS_COLOR_LIGHT = {
2070      "PING_SWEEP": "dark_orange",
2071      "PORT_SCAN": "bold dark_orange",
2072      "FLOOD": "bold red",
2073      "GRE-FLOOD": "bold red",
2074      "DNS-TORTURE": "bold red",
2075      "SCAN": "dark_orange",
2076      "UDP-AMP": "purple",
2077      "BRUTE": "bold purple",
2078      "NORMAL": "dark_green",
2079  }
2080  STATUS_ICON = {
2081      "PING_SWEEP": "\U0001f6f0\ufe0f",
2082      "PORT_SCAN": "\U0001f4e1",
2083      "FLOOD": "\U0001f30a",
2084      "GRE-FLOOD": "\U0001f310",
2085      "DNS-TORTURE": "\U0001f9e8",
2086      "SCAN": "\U0001f4e1",
2087      "UDP-AMP": "\U0001f4a5",
2088      "BRUTE": "\U0001f513",
2089      "NORMAL": "\u2705",
2090  }
2091
2092
2093  class Dashboard:
2094      def __init__(self, cfg: Config):
2095          self.cfg = cfg
2096          self.history: Dict[str, Deque[float]] = defaultdict(
```

```
            lambda: deque(maxlen=cfg.history_size)
        )
        self.start_time = time.time()

    @property
    def status_palette(self):
        return STATUS_COLOR_LIGHT if self.cfg.theme == "light" else STATUS_COLOR

    @property
    def border_style_main(self) -> str:
        return "blue" if self.cfg.theme == "light" else "cyan"

    @property
    def header_title_style(self) -> str:
        return "bold blue on white" if self.cfg.theme == "light" else "bold white on blue"

    def update_history(self, stats: Dict[str, OriginStats]) -> None:
        for k, s in stats.items():
            self.history[k].append(s.pkts_rate)
        for k in list(self.history.keys()):
            if k not in stats:
                self.history[k].append(0.0)
                if all(v == 0 for v in self.history[k]):
                    del self.history[k]

    def render(
        self,
        stats: Dict[str, OriginStats],
        dst_stats: Dict[str, DestinationStats],
        manager: MitigationManager,
        onos: OnosClient,
        total_flows: int,
    ) -> Layout:
        layout = Layout()
        layout.split_column(
            Layout(name="header", size=3),
            Layout(name="body"),
            Layout(name="footer", size=3),
        )
        layout["body"].split_row(
            Layout(name="main", ratio=2),
            Layout(name="side", ratio=1),
        )
        layout["main"].split_column(
            Layout(name="sources", ratio=2),
            Layout(name="targets", ratio=1),
        )
        layout["side"].split_column(
            Layout(name="blocks"),
            Layout(name="monitor"),
            Layout(name="actions"),
        )

        layout["header"].update(self._header(onos, total_flows, stats))
        layout["sources"].update(self._top_sources(stats))
        layout["targets"].update(self._top_targets(dst_stats))
        layout["blocks"].update(self._active_blocks(manager))
        layout["monitor"].update(self._block_monitor(manager))
        layout["actions"].update(self._recent_actions(manager))
        layout["footer"].update(self._footer())
```

```
2157          return layout
2158
2159      def _top_targets(self, dst_stats: Dict[str, DestinationStats]) -> Panel:
2160          table = Table(
2161              show_header=True, header_style="bold magenta", expand=True, padding=(0, 1)
2162          )
2163          table.add_column("Destino", no_wrap=True)
2164          table.add_column("VLAN", justify="center")   # v19beta: coluna nova
2165          table.add_column("Origens", justify="right")
2166          table.add_column("\u03a3 pps", justify="right")
2167          table.add_column("\u03a3 Mbps", justify="right")
2168          table.add_column("Indicador")
2169
2170          topo = sorted(
2171              dst_stats.items(),
2172              key=lambda x: (target_sort_rank(x[1], self.cfg), -x[1].total_pkts_rate),
2173          )[:6]
2174
2175          if not topo:
2176              table.add_row("Sem alvos", "", "", "", "", "")
2177
2178          for _, d in topo:
2179              mbps = d.total_bytes_rate * 8 / 1_000_000
2180              vlan_str = d.vlan or "\u2014"
2181              if getattr(d, "is_transit", False):
2182                  destino_str = fmt_label(d.friendly_name, d.display_ip, self.cfg.show_
      ↪ friendly)
2183                  indicator = Text("\u21aa tr\u00e2nsito L3", style="dim cyan")
2184                  origens_str = str(len(d.origins))
2185              else:
2186                  distributed = classify_distributed_attack(d, self.cfg)
2187                  indicator = (
2188                      Text("\U0001f3af DDOS DISTRIBU\u00cdDO", style="bold red")
2189                      if distributed
2190                      else Text("normal", style="dim green")
2191                  )
2192                  destino_str = fmt_label(d.friendly_name, d.display_ip or d.dst, self.cfg.
      ↪ show_friendly)
2193                  eff = getattr(d, "effective_origins", None) or d.origins
2194                  origens_str = str(len(eff))
2195              table.add_row(
2196                  destino_str,
2197                  vlan_str,
2198                  origens_str,
2199                  f"{d.total_pkts_rate:.0f}",
2200                  f"{mbps:.2f}",
2201                  indicator,
2202              )
2203          return Panel(
2204              table, title="[bold magenta]Alvos Sob Ataque[/]", border_style="magenta"
2205          )
2206
2207      def _header(
2208          self, onos: OnosClient, total_flows: int, stats: Dict[str, OriginStats]
2209      ) -> Panel:
2210          uptime = int(time.time() - self.start_time)
2211          h, m, s = uptime // 3600, (uptime % 3600) // 60, uptime % 60
2212          mode_color = "green" if self.cfg.mode == "ips" else "yellow"
2213          onos_str = (
2214              f"[green]\u2713 {onos.last_call_ms:.0f}ms[/]"
```

```
            if onos.last_call_ok
            else "[red]\u2717 DOWN[/]"
        )
        prom_str = "[green]\u2713[/]" if PROMETHEUS_ENABLED else "[red]\u2717[/]"
        total_pkts = sum(s.pkts for s in stats.values())
        total_rate = sum(s.pkts_rate for s in stats.values())
        title = Text(
            "SDN Forensic & DDoS Mitigation Platform v19.3.1",
            style=self.header_title_style,
        )
        body = Text.from_markup(
            f"Modo: [{mode_color}]{self.cfg.mode.upper()}[/]  "
            f"\u2502 ONOS: {onos_str}  \u2502 Prometheus: {prom_str}  "
            f"\u2502 Uptime: [cyan]{h:02d}:{m:02d}:{s:02d}[/]  "
            f"\u2502 Origens: [cyan]{len(stats)}[/]  "
            f"\u2502 Flows: [cyan]{total_flows}[/]  "
            f"\u2502 Pkts: [cyan]{total_pkts}[/]  "
            f"\u2502 \u03a3 pps: [cyan]{total_rate:.0f}[/]"
        )
        return Panel(body, title=title, border_style="blue")

    def _top_sources(self, stats: Dict[str, OriginStats]) -> Panel:
        table = Table(
            show_header=True, header_style="bold cyan", expand=True, padding=(0, 1)
        )
        table.add_column("Host / IP", style="white", no_wrap=True)  # v18
        table.add_column("MAC", style="dim", no_wrap=True)
        table.add_column("VLAN", justify="center")
        table.add_column("Vendor", style="dim cyan")
        table.add_column("pps", justify="right")
        table.add_column("Mbps", justify="right")
        table.add_column("Flows", justify="right")
        table.add_column("Dests", justify="right")
        table.add_column("Trend (60s)", style="cyan")
        table.add_column("Status")

        topo = sorted(
            stats.items(), key=lambda x: x[1].pkts_rate, reverse=True
        )[: self.cfg.dashboard_top_n]

        if not topo:
            table.add_row("Nenhum tr\u00e1fego", "", "", "", "", "", "", "", "", "")

        for key, s in topo:
            status = get_status(s, self.cfg)
            color = self.status_palette.get(status, "white")
            icon = STATUS_ICON.get(status, "")
            ident = fmt_label(s.friendly_name, s.ip_src or s.mac_src, self.cfg.show_
↪ friendly)
            mac_str = s.mac_src or "\u2014"
            if getattr(s, "l3_rewritten", False):
                mac_str = f"{mac_str} \u2192L3"
            rate_style = "bold red" if s.pkts_rate > 1000 else "white"
            mbps = s.bytes_rate * 8 / 1_000_000
            spark = sparkline(list(self.history.get(key, [])), width=20)
            vlan_str = s.vlan or "\u2014"
            vendor_str = (s.vendor or "\u2014")[:12]
            table.add_row(
                ident, mac_str, vlan_str, vendor_str,
                Text(f"{s.pkts_rate:.1f}", style=rate_style),
```

```
                f"{mbps:.2f}",
                str(s.flows),
                str(len(s.destinations)),
                spark,
                Text(f"{icon} {status}", style=color),
            )
        return Panel(
            table,
            title="[bold]Top Origens[/]  [dim](\u2192L3 = origem tambem roteada inter-VLAN)
↪ [/]",
            border_style="cyan",
        )

    def _active_blocks(self, manager: "MitigationManager") -> Panel:
        ids = self.cfg.mode != "ips"
        titulo = "Alertas Ativos" if ids else "Bloqueios Ativos"
        vazio = "Sem alertas ativos" if ids else "Sem bloqueios ativos"
        col_ts = "Detectado em" if ids else "Aplicada em"
        if not manager.active_blocks:
            content = Text(vazio, style="dim green")
        else:
            table = Table(show_header=True, header_style="bold red", expand=True)
            table.add_column("Origem", no_wrap=True)
            table.add_column("Tipo", no_wrap=True)
            table.add_column(col_ts, no_wrap=True)
            table.add_column("H\u00e1", no_wrap=True, justify="right")
            now = time.time()
            for key, state in list(manager.active_blocks.items())[:12]:
                age = int(now - state.blocked_at)
                age_str = f"{age//60}m{age%60:02d}s" if age >= 60 else f"{age}s"
                ts_str = datetime.datetime.fromtimestamp(state.blocked_at).strftime("%H:%M
↪ :%S")
                origem = fmt_label(state.friendly_name, key, self.cfg.show_friendly)  #
↪ v18
                table.add_row(origem, state.attack_label, ts_str, age_str)
            content = table
        return Panel(content, title=f"[bold red]{titulo}[/]", border_style="red")

    def _block_monitor(self, manager: "MitigationManager") -> Panel:
        ids = self.cfg.mode != "ips"
        titulo = "Monitora\u00e7\u00e3o de Alertas" if ids else "Monitora\u00e7\u00e3o de
↪ Bloqueios"
        if not manager.active_blocks:
            vazio = "Nenhum alerta para monitorar" if ids else "Nenhum bloqueio para
↪ monitorar"
            content = Text(vazio, style="dim green")
            return Panel(
                content,
                title=f"[bold yellow]{titulo}[/]",
                border_style="yellow",
            )
        table = Table(show_header=True, header_style="bold yellow", expand=True, padding
↪ =(0, 1))
        table.add_column("Origem", no_wrap=True)
        table.add_column("Pkts Vistos" if ids else "Pkts Dropados", justify="right")
        table.add_column("Taxa (pps)", justify="right")
        table.add_column("Atividade", no_wrap=True)
        thr = manager.cfg.unblock_residual_threshold
        interval = manager.cfg.interval if manager.cfg.interval > 0 else 1.0
        pps_thr = thr / interval
```

```
        for key, state in list(manager.active_blocks.items())[:12]:
            if ids:
                pps = state.observed_pps
                total_str = "\u2014"   # nao ha pacotes dropados em IDS
                if pps >= manager.cfg.flood_min_pkts_rate:
                    atividade = Text("\U0001f534 ATACANDO", style="bold red")
                elif pps > manager.cfg.zero_rate_threshold:
                    atividade = Text("\U0001f7e2 observando", style="green")
                else:
                    atividade = Text("\u26aa normalizando", style="dim")
            else:
                total = state.drop_pkts_total
                pps = state.drop_pps_ema
                if state.last_drop_pkts is None:
                    atividade = Text("\u23f3 medindo\u2026", style="dim")
                elif pps > pps_thr:
                    atividade = Text("\U0001f534 ATACANDO", style="bold red")
                elif pps >= 0.3:
                    atividade = Text("\U0001f7e2 monitorando", style="green")
                else:
                    atividade = Text("\u26aa ocioso", style="dim")
                total_str = f"{total:,}".replace(",", ".") if total else "0"
            origem = fmt_label(state.friendly_name, key, self.cfg.show_friendly)  # v18
            table.add_row(origem, total_str, f"{pps:,.0f}".replace(",", "."), atividade)
        return Panel(
            table,
            title=f"[bold yellow]{titulo}[/]",
            border_style="yellow",
        )

    def _recent_actions(self, manager: "MitigationManager") -> Panel:
        if not manager.recent_actions:
            content = Text("Nenhuma a\u00e7\u00e3o registrada", style="dim")
        else:
            lines = []
            for action in list(manager.recent_actions)[-8:]:
                style = "white"
                if "[BLOQUEIO]" in action or "[ATAQUE]" in action:
                    style = "red"
                elif "[ESCALADA]" in action:
                    style = "bold red"
                elif "[UNBLOCK]" in action:
                    style = "green"
                elif "[ALERTA-IDS]" in action:
                    style = "yellow"
                elif "[SKIP" in action or "[RATE" in action:
                    style = "cyan"
                lines.append(Text(action, style=style))
            content = Text("\n").join(lines)
        return Panel(content, title="[bold]A\u00e7\u00f5es Recentes[/]", border_style="
    ↪ yellow")

    def _footer(self) -> Panel:
        return Panel(
            Text.from_markup(
                "[bold]Ctrl+C[/] para sair  "
                "\u2502 Logs em [cyan]detector.log[/]  "
                "\u2502 M\u00e9tricas Prometheus em [cyan]:8000/metrics[/]"
            ),
            border_style="blue",
```

```
        )


# ============================================================================
# 11. MAIN LOOP
# ============================================================================
def run(cfg: Config) -> None:
    onos = OnosClient(cfg)
    prom = PrometheusExporter(cfg.prometheus_port) if PROMETHEUS_ENABLED else None
    manager = MitigationManager(cfg, onos, prom)
    dashboard = Dashboard(cfg) if RICH_ENABLED else None

    if cfg.reconcile_on_boot:
        try:
            n_orphans = manager.reconcile_orphans()
            if n_orphans:
                log.warning("Boot: %d regra(s) orfa(s) removida(s) na reconciliacao", n_
    ↪ orphans)
        except Exception:
            log.exception("Falha na reconciliacao de boot")

    previous: Dict[str, Dict[str, int]] = {}
    previous_dst: Dict[str, Dict[str, int]] = {}  # v18: contadores por destino

    if not RICH_ENABLED:
        log.warning("rich nao instalado - modo headless")

    def one_cycle():
        cyc_t0 = time.perf_counter()    # instrum: relogio de parede do ciclo
        onos.call_log.clear()           # instrum: zera contabilidade do ciclo
        onos.clear_cycle_cache()        # v19.2: 1 GET /hosts por ciclo (reuso intra-ciclo)
        hosts, flows = onos.get_hosts_and_flows_parallel()
        friendly = onos.get_friendly_names()                          # v18
        gateway_macs = onos.get_gateway_macs(ttl=cfg.topology_cache_ttl)  # v18
        gateway_faces = onos.get_gateway_faces(ttl=cfg.topology_cache_ttl)  # v19beta
        stats = build_statistics(
            flows, hosts, previous, cfg.interval, cfg, gateway_macs, friendly
        )
        dst_stats = build_destination_statistics(                     # v18/v19beta
            flows, hosts, previous_dst, cfg.interval, cfg, gateway_macs, friendly,
            gateway_faces,
        )
        if prom:
            prom.update(stats, cfg)
        manager.step(stats, dst_stats, flows=flows)
        impact = compute_target_impact(dst_stats)
        manager.last_target_impact = impact
        if impact.efficacy_estimate is not None:
            log.debug(
                "IMPACTO alvos=%d entregue_rate=%.0f transito_rate=%.0f eficacia~=%.1f%%",
                len(impact.per_target),
                sum(t["rate"] for t in impact.per_target.values()),
                impact.transit_rate,
                impact.efficacy_estimate * 100,
            )
        if dashboard:
            dashboard.update_history(stats)
        cyc_ms = (time.perf_counter() - cyc_t0) * 1000
        agg: Dict[str, List[float]] = defaultdict(list)
        sizes: Dict[str, int] = {}
```

```
        for ep, ms, n in onos.call_log:
            agg[ep].append(ms)
            if n >= 0:
                sizes[ep] = max(sizes.get(ep, 0), n)
        partes = ", ".join(
            f"{ep} x{len(v)}={sum(v):.0f}ms"
            + (f"(itens={sizes[ep]})" if ep in sizes else "(FALHA)")
            for ep, v in sorted(agg.items(), key=lambda kv: -sum(kv[1]))
        )
        log.info(
            "[CICLO] dur=%.0fms flows=%d origens=%d bloqueios=%d | onos: %s",
            cyc_ms, len(flows), len(stats), len(manager.active_blocks),
            partes or "(sem chamadas)",
        )
        return stats, dst_stats, len(flows)

    def update_previous(stats):
        nonlocal previous
        previous = {
            k: {
                "pkts": s.pkts,
                "bytes": s.bytes,
                "pkts_rate": s.pkts_rate,
                "bytes_rate": s.bytes_rate,
            }
            for k, s in stats.items()
        }

    if dashboard:
        console = Console()
        import termios as _termios
        import tty as _tty
        _old_term = None
        try:
            _old_term = _termios.tcgetattr(sys.stdin)
            _tty.setcbreak(sys.stdin.fileno())
        except Exception:
            log.debug("Nao foi possivel ativar cbreak mode (sem TTY?)")

        current_layout = dashboard.render({}, {}, manager, onos, 0)
        with Live(
            current_layout,
            refresh_per_second=2,
            screen=True,
            console=console,
        ) as live:
            should_exit = False
            while not should_exit:
                try:
                    stats, dst_stats, total_flows = one_cycle()
                    update_previous(stats)
                    current_layout = dashboard.render(
                        stats, dst_stats, manager, onos, total_flows
                    )
                    live.update(current_layout)
                    elapsed = 0.0
                    while elapsed < cfg.interval:
                        key = _try_read_key(timeout=0.2)
                        elapsed += 0.2
                        if key and key.lower() == cfg.menu_key.lower():
```

```
                            live.stop()
                            should_exit = show_menu(
                                cfg, manager, onos, console, current_layout
                            )
                            try:
                                _tty.setcbreak(sys.stdin.fileno())
                            except Exception:
                                pass
                            if not should_exit:
                                live.start()
                            break
                        if key and key.lower() == "q":
                            should_exit = True
                            break
                except KeyboardInterrupt:
                    should_exit = True
                except Exception:
                    log.exception("Erro no ciclo principal")
                    time.sleep(cfg.interval)
        if _old_term is not None:
            try:
                _termios.tcsetattr(sys.stdin, _termios.TCSADRAIN, _old_term)
            except Exception:
                pass
    else:
        while True:
            try:
                stats, dst_stats, total_flows = one_cycle()
                update_previous(stats)
                attacked = sum(
                    1 for d in dst_stats.values() if classify_distributed_attack(d, cfg)
                )
                print(
                    f"[{datetime.datetime.now():%H:%M:%S}] "
                    f"origens={len(stats)} flows={total_flows} "
                    f"bloqueios={len(manager.active_blocks)} "
                    f"alvos_ddos_distrib={attacked}"
                )
                time.sleep(cfg.interval)
            except KeyboardInterrupt:
                break
            except Exception:
                log.exception("Erro no ciclo principal")
                time.sleep(cfg.interval)


# ======================================================================
# 11.5. HELPERS: menu, export snapshot, key handling
# ======================================================================
def _try_read_key(timeout=0.0):
    try:
        import select as _sel
        if _sel.select([sys.stdin], [], [], timeout)[0]:
            return sys.stdin.read(1)
    except Exception:
        pass
    return None


def export_snapshot(console, layout, title=""):
```

```
    if not RICH_ENABLED or console is None or layout is None:
        return []
    from rich.console import Console as _C
    from rich.panel import Panel as _P
    from rich.text import Text as _T
    ts = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    files = []
    recorder = _C(record=True, width=200)
    if title:
        recorder.print(_P(_T(title, style="bold"), border_style="blue"))
    recorder.print(layout)
    try:
        html_path = f"snapshot_{ts}.html"
        recorder.save_html(html_path, inline_styles=True)
        files.append(html_path)
    except Exception:
        log.exception("Falha ao exportar HTML")
    try:
        svg_path = f"snapshot_{ts}.svg"
        recorder.save_svg(svg_path, title=title or "SDN DDoS Detector")
        files.append(svg_path)
    except Exception:
        log.exception("Falha ao exportar SVG")
    return files


def show_menu(cfg, manager, onos, console=None, layout=None):
    while True:
        print("\n" + "=" * 60)
        print("  MENU DO DETECTOR v19.3.1")
        print("=" * 60)
        print(f"  Modo: {cfg.mode.upper():>3s}  |  "
              f"Bloqueios ativos: {len(manager.active_blocks)}  |  "
              f"Tema: {cfg.theme}")
        print("-" * 60)
        print("  [1] Alternar IPS <-> IDS")
        print("  [2] Limpar TODOS os bloqueios (ONOS + estado)")
        print("  [3] Limpar somente state.json")
        print("  [4] Listar regras de mitigacao no ONOS")
        print("  [5] Exportar snapshot (HTML + SVG)")
        print("  [6] Exportar snapshot estilo DISSERTACAO")
        print("  [7] Alternar tema dark <-> light")
        print("  [V] Voltar pro dashboard")
        print("  [Q] Sair do detector")
        print("=" * 60)
        op = input("  > ").strip().lower()

        if op == "1":
            old_mode = cfg.mode
            cfg.mode = "ids" if cfg.mode == "ips" else "ips"
            print(f"  > Modo alterado: {old_mode.upper()} -> {cfg.mode.upper()}")
            if old_mode == "ids" and cfg.mode == "ips":
                promoted = 0
                for origin_key, state in list(manager.active_blocks.items()):
                    if not state.flow_ids:
                        flow_ids = manager._drop(
                            state.target_ip,
                            state.target_mac,
                            is_brute=False,
                        )
```

```
                        if flow_ids:
                            state.flow_ids = flow_ids
                            promoted += 1
                if promoted:
                    print(f"  > {promoted} alerta(s) promovido(s) a bloqueio(s) reais no
↪ ONOS")
                manager._save_state()
            elif old_mode == "ips" and cfg.mode == "ids":
                removed = 0
                for origin_key, state in list(manager.active_blocks.items()):
                    if state.flow_ids and (state.target_ip or state.target_mac):
                        n = manager.onos.delete_mitigation_flows_for(
                            state.target_ip, state.target_mac
                        )
                        state.flow_ids = []
                        removed += n
                if removed:
                    print(f"  > {removed} regra(s) removida(s) do ONOS (modo IDS nao
↪ bloqueia)")
                manager._save_state()
            time.sleep(2)
        elif op == "2":
            print("  Removendo regras do ONOS...")
            flows = onos.get_flows()
            n = 0
            for f in flows:
                if f.get("priority") != 40000:
                    continue
                instructions = f.get("treatment", {}).get("instructions", [])
                is_drop = (not instructions) or all(
                    i.get("type") == "NOACTION" for i in instructions
                )
                if not is_drop:
                    continue
                crits = {c.get("type") for c in f.get("selector", {}).get("criteria", [])}
                if "IPV4_SRC" not in crits and "ETH_SRC" not in crits:
                    continue
                if onos.delete_flow(f["deviceId"], f["id"]):
                    n += 1
            manager.active_blocks.clear()
            manager._save_state()
            print(f"  > {n} regra(s) removida(s) + estado interno limpo")
            time.sleep(2)
        elif op == "3":
            p = Path(cfg.state_filename)
            if p.exists():
                p.unlink()
            manager.active_blocks.clear()
            print("  > state.json removido + active_blocks zerado")
            time.sleep(2)
        elif op == "4":
            flows = onos.get_flows()
            n = 0
            print()
            for f in flows:
                if f.get("priority") != 40000:
                    continue
                instructions = f.get("treatment", {}).get("instructions", [])
                is_drop = (not instructions) or all(
                    i.get("type") == "NOACTION" for i in instructions
```

```
                )
                if not is_drop:
                    continue
                crits = f.get("selector", {}).get("criteria", [])
                types = {c.get("type") for c in crits}
                if "IPV4_SRC" not in types and "ETH_SRC" not in types:
                    continue
                src = next(
                    (c.get("ip") or c.get("mac") for c in crits
                     if c.get("type") in ("IPV4_SRC", "ETH_SRC")),
                    "?",
                )
                print(f"  {src:25s} state={f.get('state','?'):10s} "
                      f"pkts={f.get('packets', 0):>10}")
                n += 1
            if n == 0:
                print("  (nenhuma regra de mitigacao ativa)")
            input("\n  ENTER para voltar...")
        elif op == "5":
            files = export_snapshot(console, layout)
            for f in files:
                print(f"  > {f}")
            if not files:
                print("  > Falha ao exportar")
            time.sleep(2)
        elif op == "6":
            titulo = input("  Titulo da figura: ").strip()
            if not titulo:
                titulo = f"Detector SDN v19.3.1 - {datetime.datetime.now():%Y-%m-%d %H:%M
    ↪ :%S}"
            files = export_snapshot(console, layout, title=titulo)
            for f in files:
                print(f"  > {f}")
            time.sleep(2)
        elif op == "7":
            cfg.theme = "light" if cfg.theme == "dark" else "dark"
            print(f"  > Tema alterado para {cfg.theme}")
            time.sleep(2)
        elif op == "v" or op == "":
            return False
        elif op == "q" or op == "0":
            return True
        else:
            print(f"  ? Opcao invalida: {op}")
            time.sleep(1)


# ============================================================================
# 12. ENTRY POINT
# ============================================================================
def parse_args():
    p = argparse.ArgumentParser(description="SDN DDoS Detector v19.3.1")
    p.add_argument("--config", default="config.yaml")
    p.add_argument("--mode", choices=["ips", "ids"])
    p.add_argument("--theme", choices=["dark", "light"])
    p.add_argument("--block-strategy", choices=["ip", "mac", "auto"],
                   help="ancora da regra de DROP: ip|mac|auto")
    p.add_argument("--debug", action="store_true")
    return p.parse_args()

```

```
2743
2744 def main():
2745     global log
2746     args = parse_args()
2747     cfg = Config.load(args.config)
2748     if args.mode:
2749         cfg.mode = args.mode
2750     if args.theme:
2751         cfg.theme = args.theme
2752     if args.block_strategy:
2753         cfg.block_strategy = args.block_strategy
2754     log = setup_logging(cfg.log_filename, logging.DEBUG if args.debug else logging.INFO)
2755     log.info("=" * 60)
2756     log.info("Iniciando detector v19.3.1 (modo=%s, ancora=%s)", cfg.mode.upper(), cfg.
      ↪ block_strategy)
2757     log.info("ONOS: %s", cfg.onos_url)
2758     log.info("Allowlist IPs: %s", cfg.allowlist_ips or "(vazia)")
2759     log.info("Allowlist MACs: %s", cfg.allowlist_macs or "(vazia)")
2760     log.info("UNBLOCK por contador de DROP: %s (limiar residual: %d pkts/ciclo)",
2761              cfg.unblock_by_drop_counter, cfg.unblock_residual_threshold)
2762     log.info("Transit detection: %s | Auto-router: %s | Skip-ICMP-scan: %s",
2763              cfg.transit_detection_enabled, cfg.auto_detect_routers, cfg.skip_icmp_scan)
2764     log.info("=" * 60)
2765     print(f"Iniciando detector v19.3.1 (modo={cfg.mode.upper()}, ancora={cfg.block_
      ↪ strategy})...", file=sys.stderr)
2766     print("  Aperte 'M' para abrir o menu | 'Q' para sair", file=sys.stderr)
2767     try:
2768         run(cfg)
2769     except Exception:
2770         log.exception("Erro fatal")
2771         print(traceback.format_exc(), file=sys.stderr)
2772         sys.exit(1)
2773     finally:
2774         log.info("Detector encerrado")
2775
2776
2777 if __name__ == "__main__":
2778     main()
```

C. ARQUIVO DE CONFIGURAÇÃO: DDSO V19.3.1

Este apêndice reproduz o arquivo `config.yaml` empregado nos ensaios do Capítulo 4. Os valores listados são os efetivamente utilizados nas campanhas e incluem os limiares heurísticos de detecção, a *allowlist* de endereços de infraestrutura protegidos contra bloqueio e os parâmetros de operação do orquestrador, entre eles o limiar de inundação (`flood_min_pkts_rate = 10000`) e o período do ciclo de análise (`interval = 1 s`) discutidos na Seção 3.4. Parâmetros não declarados no arquivo assumem o valor padrão do código; variáveis de ambiente têm precedência sobre o YAML.

Listagem C.1: Arquivo de configuração `config.yaml` do DDSO v19.3.1.

```
# ============================================================
# DDSO v19.3.1 - arquivo de configuracao
# ============================================================
# Tudo e' opcional: o que nao estiver aqui usa o default do codigo.
# Variaveis de ambiente (ONOS_URL, ONOS_USER, ONOS_PASS) tem
# prioridade sobre o YAML.
# ============================================================
# --- ONOS ---
onos_url: "http://127.0.0.1:8181/onos/v1"
onos_user: "onos"
onos_pass: "rocks"
onos_timeout: 5.0
# --- Operacao ---
mode: "ids"               # "ips" (mitigacao ativa) ou "ids" (so monitora).
                          # Nos experimentos, sobreponha via --mode ips / --mode
    ↪ ids
                          # (para a condicao OFF, o DDSO fica desligado).
interval: 1.0             # periodo do ciclo de analise, em segundos.
                          # O flowPollFrequency do provedor OpenFlow no ONOS foi
                          # fixado em 1 s (constante metodologica; ver Cap. 3), de
                          # modo que os contadores tornam-se visiveis a cada 1 s e
    ↪  a
                          # latencia de deteccao acompanha este ciclo, e nao o
                          # default de ~5 s do ONOS. O burst detection (abaixo)
    ↪ ainda
                          # oferece um caminho rapido para rajadas obvias.
prometheus_port: 8000
csv_filename: "mitigacao_forense_log.csv"
state_filename: "detector_state.json"
log_filename: "detector.log"
# --- Heuristicas de SCAN ---
scan_min_destinations: 5
scan_min_flows: 5
scan_max_pkts_avg: 200
# --- Heuristicas de BRUTE FORCE ---
```

```
brute_min_conns: 20
brute_ports:
  - 22       # SSH
  - 23       # Telnet
  - 2323     # Telnet alternativo (Mirai)
  - 80       # HTTP admin
  - 8080
  - 7547     # TR-069 (Mirai)
  - 37215    # Huawei (Mirai)
# --- Heuristicas de FLOOD ---
flood_min_pkts_rate: 10000      # pps minimo para classificar FLOOD
flood_min_bytes_rate: 10000000  # OU bytes/s minimo (10 MB/s)
flood_max_destinations: 3       # flood concentra em poucos alvos
# --- Heuristicas de UDP AMPLIFICATION ---
amp_ports:
  - 53      # DNS
  - 123     # NTP
  - 1900    # SSDP
amp_min_avg_packet_size: 500
amp_min_pkts_rate: 100   # evita falso positivo em DNS legitimo
# --- DNS Water Torture (Mirai ATK_VEC_DNS) ---
dns_torture_min_queries: 500
dns_torture_min_rate: 100
# --- GRE Flood (Mirai ATK_VEC_GREIP/GREETH, IP protocol 47) ---
gre_min_pkts_rate: 5000
# --- Ataque DDoS distribuido (varios bots, mesmo destino) ---
dest_distributed_min_origins: 3
dest_distributed_min_total_pps: 5000
# --- Mitigacao ---
max_mitigations_per_minute: 20
zero_rate_threshold: 5.0        # histerese: tolera ruido de fundo (~1 pps)
mitigation_zero_rate_cycles: 35 # 35 ciclos x 1 s = 35 s calmos antes do UNBLOCK
# --- Allowlist (NUNCA bloqueia esses) ---
allowlist_ips:
  - "10.0.0.10"          # servidor de monitoramento
  - "164.72.30.2"        # OPNsense (firewall/router central - NUNCA bloquear!)
  - "164.72.30.1"        # backbone do OPNsense
  - "164.72.22.254"      # gateway vlan 22 BOT
  - "164.72.55.254"      # gateway vlan 55 Target
  - "164.72.12.254"      # gateway vlan 12 DNS
  - "172.16.23.254"      # gateway vlan 23 CNC
  - "172.16.23.1"        # gateway vlan 23 CNC
  - "172.16.200.254"     # gateway vlan 200 NOC
  - "192.168.1.1"        # gateway vlan gerencia
  - "164.72.200.254"     # gateway vlan NOC
allowlist_macs: []
# --- Deteccao de nos de transito (anti-falso-positivo) ---
transit_detection_enabled: true
transit_threshold_pps: 1000.0
auto_detect_routers: true
topology_cache_ttl: 60.0
skip_icmp_scan: true
```

```
# --- Burst detection ---
burst_detection_enabled: true
burst_multiplier: 1.5           # 10000 x 1.5 = 15000 pps -> bypassa o EMA
# --- Dashboard ---
theme: "dark"
menu_key: "m"
history_size: 150
dashboard_top_n: 8
```

D. REGISTRO DE AUDITORIA DAS EXECUÇÕES

Este apêndice documenta as execuções excluídas das campanhas estatísticas e o critério aplicado a cada uma, em atendimento à rastreabilidade exigida de um estudo experimental. Os critérios de validade foram estabelecidos antes da análise dos resultados e dizem respeito à validade da captura, e não ao valor de exposição medido, de modo que a exclusão não é condicionada ao desfecho da repetição. São dois: a confirmação de que o ataque efetivamente atingiu o alvo, verificada pela presença de amostras rotuladas `FLOOD` na telemetria da vítima; e a ausência de sobreposição de janelas de disparo entre repetições consecutivas, verificada pela coerência entre a duração observada e a duração nominal do disparo. Cada condição foi executada trinta vezes, à exceção da estratégia *mac* do vetor SYN, cuja coleta registrou uma captura adicional. A Tabela D.1 relaciona as capturas excluídas.

Tabela D.1: Capturas excluídas das campanhas estatísticas, por vetor e estratégia, com o critério de validade não atendido e a origem provável da falha. As condições não listadas mantiveram as trinta repetições.

Vetor	**Estratégia**	**Execução**	**Exp. (s)**	**Critério não atendido**	**Origem**
SYN	off	01	58	Sobreposição de janela de disparo (exposição $\approx 2\times$ a mediana)	Coleta / gerador
SYN	off	09	57	Sobreposição de janela de disparo	Coleta / gerador
SYN	off	11	54	Sobreposição de janela de disparo	Coleta / gerador
SYN	off	29	54	Sobreposição de janela de disparo	Coleta / gerador
SYN	auto	—	—	Registro inválido na coleta	Coleta
SYN	mac	(extra)	—	Repetição adicional (31.ª captura)	Coleta
UDP	off	24	0	Ataque não atingiu o alvo (sem amostras `FLOOD`)	Gerador / rede

Após as exclusões, os tamanhos amostrais efetivos são de vinte e seis repetições na condição de referência do vetor SYN, vinte e nove na estratégia adaptativa do mesmo vetor, vinte e nove na condição de referência do vetor UDP e trinta nas demais condições dos quatro vetores. Nenhuma exclusão decorreu de falha atribuível ao controlador ou ao comutador sob avaliação; as ocorrências registradas originaram-se na etapa de geração de tráfego ou na coleta de telemetria. Registra-se, por transparência, que capturas com valores atípicos, porém válidas quanto aos dois critérios, foram mantidas na amostra, como a repetição de número treze da condição de referência do vetor ACK, cuja exposição de trinta e dois segundos, inferior à das demais repetições, foi conservada por corresponder a uma medição legítima.