Capítulo 4

Protegendo Redes de Computadores na era do Plano de Dados Programáveis

Arthur Selle Jacobs[1], Antônio João Gonçalves de Azambuja[2], Alberto Egon Schaeffer-Filho[3], Jéferson Campos Nobre[4], Juliano Araújo Wickboldt[5], Lisandro Zambenedetti Granville[6], Luciano Paschoal Gaspary[7], Weverton Luis da Costa Cordeiro[8]

Instituto de Informática (INF)
Universidade Federal do Rio Grande do Sul (UFRGS)

Abstract

Recent advances in Software Defined Networking (SDN) have expanded our ability to program the network towards the data plane. Through domain-specific languages like P4, network operators can quickly implement new protocols in forwarding devices, customize their functionality and develop innovative services. This flexibility comes, however, at a cost: security and correctness properties of the entire network (e.g., isolation and accessibility) become much more difficult to guarantee, because the behavior of the network is now determined by a combination of the configuration maintained by the control plane and data plane programs residing on forwarding devices. In this context, existing tools for network verification, which depend on a fixed and invariant model of the data plane, are inadequate for programmable data planes. This chapter aims to foster discussion on this subject, by presenting (i) how the concept of data plane programmability can be used to make computer networks more secure, and (ii) what major security challenges emerge along with the concept.

[1]E-mail: asjacobs@inf.ufrgs.br
[2]E-mail: azambuja@inf.ufrgs.br
[3]E-mail: alberto@inf.ufrgs.br
[4]E-mail: jeferson.nobre@inf.ufrgs.br
[5]E-mail: jwickboldt@inf.ufrgs.br
[6]E-mail: granville@inf.ufrgs.br
[7]E-mail: paschoal@inf.ufrgs.br
[8]E-mail: weverton.cordeiro@inf.ufrgs.br

Resumo

Os avanços recentes em Redes Definidas por Software (Software Defined Networking, SDN) expandiram nossa capacidade de programar a rede em direção ao plano de dados. Através de linguagens específicas de domínio como o P4, os operadores de rede podem rapidamente implementar novos protocolos em dispositivos de encaminhamento, personalizar suas funcionalidades e desenvolver serviços inovadores. Essa flexibilidade vem, no entanto, com um custo: as propriedades de segurança e de corretude em toda a rede (e.g., isolamento e acessibilidade) tornam-se muito mais difíceis de garantir, porque o comportamento da rede agora é determinado por uma combinação da configuração mantida pelo plano de controle e os programas do plano de dados que residem nos dispositivos de encaminhamento. Neste contexto, as ferramentas existentes para verificação de redes, as quais dependem de um modelo fixo e invariante do plano de dados, são inadequadas para planos de dados programáveis. Este capítulo visa fomentar discussão sobre esse assunto, ao apresentar (i) como o conceito de programabilidade do plano de dados pode ser usado para tornar as redes de computadores mais seguras, e (ii) quais os principais desafios de segurança que emergem juntamente com o conceito.

4.1. Introdução

Os avanços recentes em Redes Definidas por *Software* (*Software Defined Networking*, SDN) expandiram a capacidade de programar a rede em direção ao plano de dados. Segundo a *Open Networking Foundation* (OFN), as SDN são arquiteturas dinâmicas, gerenciáveis com adaptabilidade, que permitem a separação entre o controle da rede e as funções de encaminhamento. Com SDN, as infraestruturas passaram a ser mais flexíveis, logicamente centralizadas, e com uma *interface* aberta e bem definida. Através de linguagens específicas de domínio, os operadores de rede podem rapidamente implementar novos protocolos em dispositivos de encaminhamento, personalizar suas funcionalidades e desenvolver serviços inovadores.

O protocolo *OpenFlow* (OF) tem papel fundamental na habilitação das arquiteturas SDN, estabelecendo uma camada de abstração da rede física para o elemento de controle, permitindo a configuração e/ou manipulação do plano de dados da rede por meio de uma programação via *software* [McKeown et al. 2008, Garcia et al. 2018, Feferman et al. 2018]. Esse protocolo foi desenvolvido em um projeto *Open Source* no contexto de pesquisas realizadas na *Stanford University* e *University of California - Berkeley*. O protocolo OF permite que dispositivos de fabricantes diferentes suportem novas funções e protocolos, habilitando uma separação do plano de controle e dados. O plano de controle externo e centralizado permite a programação de encaminhamento unificado, políticas e suportes de segurança na criação de redes flexíveis e reconfiguráveis. A Figura 4.1 apresenta uma visão geral de SDN implementado com *OpenFlow* [Fernandes and Rothenberg 2014, McKeown et al. 2008].

A adoção do SDN abriu espaço para avanços no cenário de rede, permitindo a reorganização lógica da rede em um plano de aplicação, considerando a inteligência da rede; um plano de controle; e um plano de dados com a manipulação do fluxo [Kreutz et al. 2014]. A Figura 4.2 apresenta uma visão geral da arquitetura SDN, com os três planos conceituais e interfaces de comunicação: 1) Plano de aplicação (*Applica-*

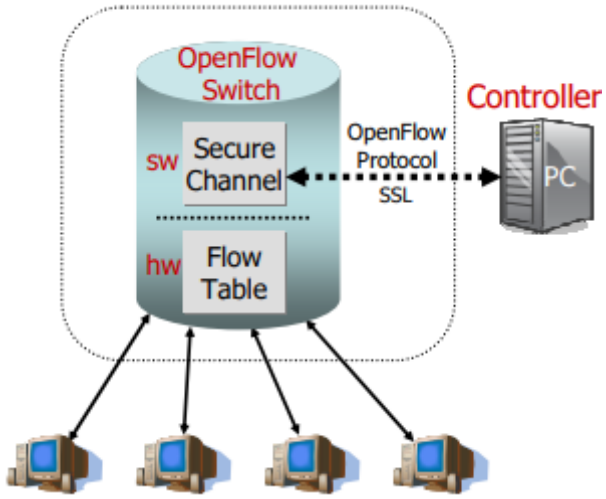

Figura 4.1: SDN implementado com OpenFlow (Fonte: [McKeown et al. 2008]).

tion plane): responsável por orquestrar serviços na malha de rede definida pelo plano de dados; 2) Plano de controle (*Control plane*): implementa abstrações de alto nível como gráfico de rede, objetivos de fluxo e intenções para aplicativos visando a configuração e gerenciamento de elementos de rede. São exemplos de controladores SDN: *ONOS*, *NOX*, *POX*, *Floodlight* e *Ryu*; e 3) Plano de dados (*Date plane*): considera os dispositivos, roteadores e/ou *switches*, que realizam encaminhamento de pacotes e processamento pela rede, como por *middleboxes* pela rede.

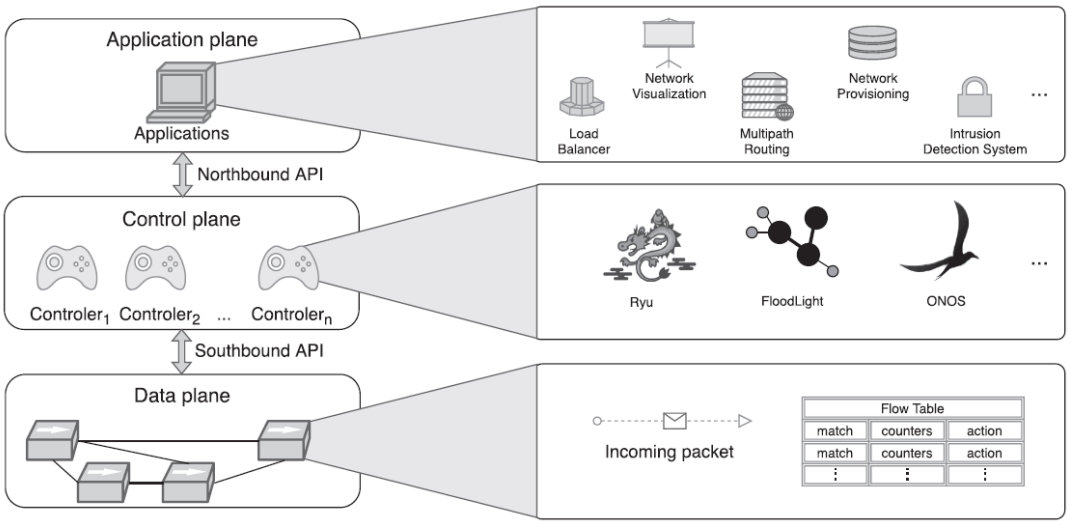

Figura 4.2: Planos conceituais SDN e interfaces de comunicação com planos de controle, aplicação e de dados (Fonte: [da Silva et al. 2015]).

Aplicações que usam o *software* do plano de encaminhamento permitem que sejam definidos quais pacotes devem ser processados e consequentemente ações devem ser tomadas. No entanto, a possibilidade de programar o plano de dados traz importantes desafios: as propriedades de segurança e de corretude em toda a rede (e.g., isolamento

e acessibilidade) tornam-se muito mais difíceis de garantir, porque o comportamento da rede agora é determinado por uma combinação da configuração mantida pelo plano de controle e os programas do plano de dados que residem nos dispositivos de encaminhamento. As ferramentas existentes para verificação de redes que dependem de um modelo fixo e invariante do plano de dados, são inadequadas para planos de dados programáveis.

O presente capítulo apresenta os principais fundamentos de programabilidade do plano de dados e P4. Tais fundamentos permitem explorar desafios de segurança relacionados essa programabilidade, assim como abordagens que a usem em mecanismos de segurança. Além disso, são discutidos exemplos e códigos-fonte para auxiliar na descrição de propriedades da utilização de programabilidade do plano de dados.

O presente capítulo está organizado da seguinte forma. Na Seção 4.2, são revisados os principais fundamentos em relação à Programabilidade do Plano de Dados e P4. Na Seção 4.3, a *Protocol-Independent Switch Architecture* (PISA) e o *pipeline* de processamento de pacotes são apresentados. Na Seção 4.4, propriedades de segurança de rede frente a aspectos de programabilidade são discutidos. Na Seção 4.5, abordagens para a verificação de políticas de segurança são apresentadas. Na Seção 4.6, mecanismos para a imposição de políticas de segurança são discutidos. Na Seção 4.7, cobre-se um conjunto de tópicos avançados sobre segurança em planos de dados programáveis. Finalmente, considerações finais e perspectivas de trabalhos futuros são apresentados na Seção 4.8.

4.2. P4 - *Programming Protocol-Independent Packet Processors*

Linguagens de programação de domínio específico, como o P4, foram propostas para especificar a lógica de processamento de pacotes de dispositivos no plano de dados programáveis por meio de uma arquitetura de alto nível independente de abstrações. Os operadores de rede podem rapidamente implementar novos protocolos em dispositivos de encaminhamento, personalizar suas funcionalidades e desenvolver serviços inovadores.

A flexibilidade advinda do plano de dados programável traz consigo um custo relacionado com as propriedade de segurança e corretude em toda a rede. As ferramentas existentes para verificação de redes, as quais dependem de um modelo fixo e invariante do plano de dados, são inadequadas para planos de dados programáveis. Diante das limitações inerentes nos ASICs *(Application Specific Integrated Circuits)* dos dispositivos de rede, que tornavam as alterações e implementações de novas funcionalidade um processo rígido e/ou inflexível, foram necessárias mudanças na forma de processamento dos pacotes.

As mudanças na forma de processamento dos pacotes têm foco para o plano de dados considerando o referido contexto em uma abordagem para descrever o gerenciamento do fluxo de dados e como fazer. Com este cenário emerge a linguagem de processamento de pacotes P4 em 2014, quando foi publicado *Programming Protocol-Independent Packet Processors* [Bosshart et al. 2014a], com uma uma gestão realizada pela organização *P4 Language Consortium* [9].

P4 [Bosshart et al. 2014b] é um DSL de alto nível para programar como os *switches* processam os pacotes. P4 é independente de destino, ou seja, adequado para descre-

[9]https://p4.org/

ver o comportamento de vários tipos de *switch* (por exemplo, ASICs de função xed, NPUs, comutadores de *software*, FPGAs). A linguagem abstrai a análise e o processamento de pacotes, fornecendo um encaminhamento generalizado. O código P4 é organizado logicamente da seguinte forma (veja Figura 4.3):

1. Declaração de dados: é uma seção que define o formato do cabeçalho do pacote e as informações de metadados que podem ser usadas para sua análise. Esta seção é mapeada em um cabeçalho e um barramento de metadados que transporta essas informações por todos os estágios de processamento. Os tipos de cabeçalho são declarados de forma semelhante às estruturas em C, ou seja, os campos são definidos em uma ordem específica e com um tamanho pré-determinado. As Figuras 4.4(a) e (b) ilustram a declaração do cabeçalho *Ethernet* padrão e de um protocolo arbitrário que utiliza *tags* como identificadores de origem e destino, respectivamente;

2. *Parser logic*: é uma seção que especifica como, quando e em que ordem cada um dos cabeçalhos deve ser analisado. Esta seção de um programa P4 é mapeada para os elementos *parser* e *deparser* do modelo de encaminhamento (veja a Figura 4.3). Esses elementos são então responsáveis por extrair o campo de cabeçalho dos pacotes em seu ingresso (*parser*) e no processo de tradução de estruturas de dados em um formato que possa ser armazenado e reconstruído na sequência no mesmo ou em um ambiente computacional diferente; e

3. *Match-action tables and control flows*: é uma seção que especifica tabelas do tipo *OpenFlow* capazes de corresponder em campos de cabeçalho arbitrários e modificação de cabeçalhos de pacotes (e metadados) por meio de ações personalizadas. Também expressa controle funções sem a ordem e circunstâncias que cada tabela deve ser executada. Esta seção é mapeada no contexto de configuração dos elementos compatíveis com *match-action* nos *pipelines* de entrada e saída. O *pipeline* de entrada executa o pacote (agnóstico de saída) e também estipula as intenções de saída (por exemplo, qual porta encaminhar um pacote). O *pipeline* de saída realiza ainda mais modificações de pacote necessárias, reescrevendo um endereço de origem de cabeçalho *Ethernet* com o endereço MAC da porta de saída.

Para [Bosshart et al. 2014b], três desafios foram estabelecidos: 1) Processamento de pacotes configuráveis: busca realizar a análise do programa para suportar a declaração de cabeçalhos, fazendo com que essa análise não fique dependente a um formato de pacote específico ou cabeçalho, tornando possível a inserção de novos cabeçalhos ao longo do tempo; 2) Flexibilização nas tabela de pacotes: busca configurar as tabelas de correspondência permitindo estruturas em série, paralelas ou até híbridas, para possibilitar inserção de novas funcionalidades no programa; e 3) Primitivas genéricas de processamento de pacotes: busca disponibilizar primitivas genéricas (por exemplo, *copy*, *add*, *remove* e *modify*) que permitam a alteração dos metadados e dados, podendo ser facilmente utilizadas.

Esses desafios têm fomentado pesquisas pela indústria e academia no contexto das tecnologias programáveis, dentre elas: 1) Switches Application Specific Integrated Circuits (ASIC): Barefoot Tofino, Cisco Doppler, Cavium (Xpliant), Intel Flexpipe; 2) Field Programmable Gate Arrays (FPGA): Altera, Xilinx; 3) Central Processing Unit (CPU):

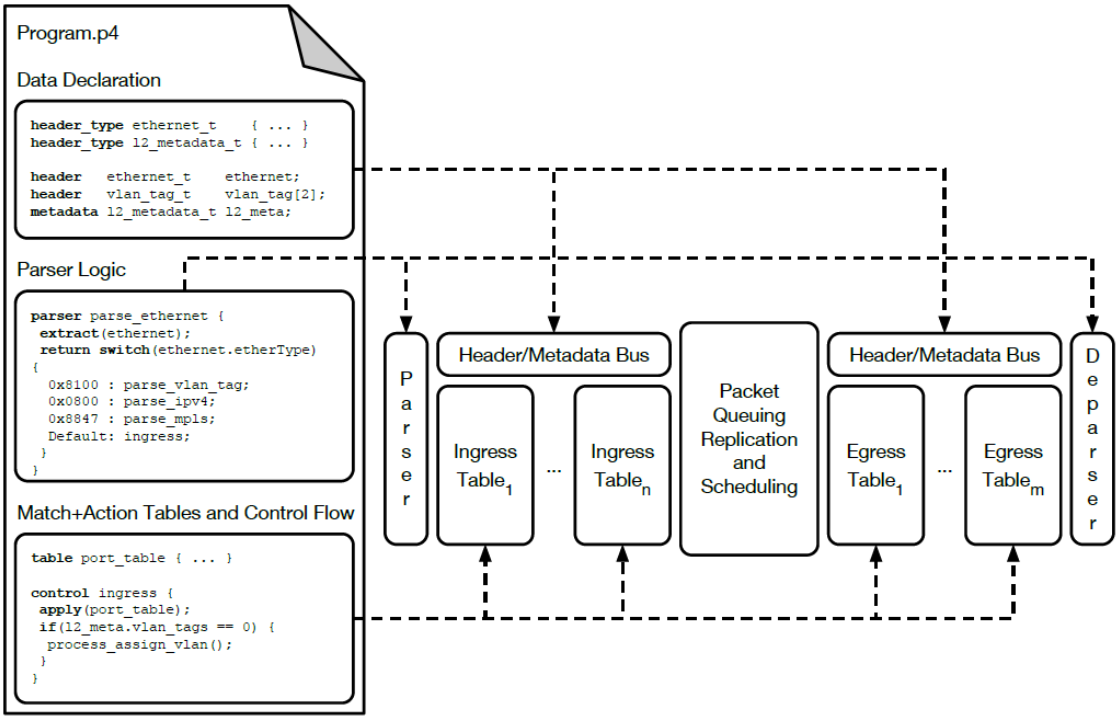

Figura 4.3: Seções de código P4 e mapeamento para o modelo de encaminhamento abstrato (Adaptado de [Kim et al. 2006]).

```
header ethernet {
  fields {
    dst_addr : 48;
    src_addr : 48;
    ether_type : 16;
  }
}
```

((a)) Ethernet header

```
header customl2proto {
  fields {
    src_tag : 16;
    dst_tag : 16;
    ethertype : 16;
  }
}
```

((b)) Custom L2 protocol header

Figura 4.4: Exemplos de declaração de cabeçalho P4.

DPDK, Open Vswitch, VPP, BPF; e 4) Network Processor Unit: Netronome, EZchip. A programação desses equipamentos tem uma complexidade gerada pelo fato de terem *interfaces* proprietárias com programação de baixo nível.

A linguagem P4, baseada na arquitetura de *switch* programável *Protocol-Independent Switch Architecture* (PISA), possibilita abstrair o plano de dados fazendo com que a complexidade da programação do comportamento dos equipamentos seja superada com uma programação de alto nível. Com o PISA insere-se um novo paradigma com *hardware* baseado em um *pipeline* de configuração e reconfiguração utilizando o modelo de *match-action*. Com isso, essa arquitetura permite maior flexibilidade sem impactar a perfomance e permite que o programador recupere o controle do plano de dados com funções aperfeiçoadas para uma rede específica.

Para os autores Garcia et al. [Garcia et al. 2018] a programabilidade do plano de dados apresenta os seguintes benefícios: 1) Controle: permite que o dispositivo de rede seja controlado da forma como planejado; 2) Capacidade: possibilita adicionar no-

vas funções aos dispositivos, já que possuem a flexibilidade na sua programação; 3) Exclusividade: implementa protocolos personalizados; 4) Eficiência: otimiza a capacidade para o uso dos recursos presentes nos dispositivos; 5) Confiabilidade: cria uma camada de proteção para não utilizar funções implementadas por terceiros; e 6) Monitoramento: acompanha o processamento e exporta dados para auferir o desempenho.

São três os objetivos definidos na linguagem P4 no contexto da programação de redes [Feferman et al. 2018]: 1) Independência de protocolos: o *switch* deve ter a capacidade para analisar qualquer tipo de protocolo, sendo que deve ser declarado inicialmente no programa como analisar o cabeçalho e as ações aplicando o modelo de *Match-action*; 2) Independência de arquitetura: o programa P4 busca uma abstração na camada física, fazendo com que a gestão das particularidades dos equipamentos seja realizada pelo próprio compilador; e 3) Reconfiguração em campo: o processamento de pacotes pode ser alterado durante a sua execução.

4.3. PISA e Pipeline de Processamento de Pacotes

A PISA (Protocol-Independent Switch Architecture), tem três partes: 1) *Parser*; 2) *Pipeline match-action*; e 3) *Deparser*. Baseado nessa arquitetura um programa P4 compreende declarações de cabeçalho (*header*), máquina de estado do analisador de pacotes (*parser*) e as tabelas de ação de correspondência (*pipeline match-action*). A linguagem P4 arca com a implementação de *parser*, o qual percorre os *headers* do começo ao fim, obtendo os valores de campo considerando o percurso. Esses valores são encaminhados para o processamento das tabelas *match-action*, identificando os campos de pacotes e metadados que serão lidos com as possíveis ações executadas como respostas. As declarações *header* determinam os nomes e as larguras dos campos para os cabeçalhos de protocolo para estabelecer o destinto de processamento do programa P4.

São os seguintes os componentes do P4 associados à arquitetura PISA: 1) *Parser* programável: busca identificar e especificar o formato para processamento, em uma cadeia de *bits*, dos cabeçalhos que estão presentes no pacote. *Parser* em sua definição determina a forma para identificar e reconhecer cabeçalhos ou sequências de cabeçalhos válidos nos pacotes; 2) *Header*: visa especificar a largura, restrições de tipos, valores e estrutura de uma série de campos de *bits* implementados pelo programador; 3) *Pipeline match-action*: tem como propósito a comparação de um ou mais campos dos cabeçalhos obtidos no *parser* com uma tabela programada pelo controlador, realizando uma correspondência de um campo e um valor que apresentam uma associação nas tabelas; 4) *Tables*: são mecanismos para realizar os processamentos dos pacotes. As tabelas são ações associadas aos pacotes conforme o *matching* executado nos *parsers*; e 5) *Actions*: conjunto de primitivas usados para uma certa função customizada, podendo incluir primitivas como por exemplo: *copy-header*, *remove-header*, *add*.

No modelo abstrato de encaminhamento PISA, apresentado na Figura 4.5, os *switches* encaminham pacotes por meio de uma analisador programável seguido por estágios de *match+action*, organizados em série. No primeiro, os pacotes entram na fase de análise ou de *parser* em que seus cabeçalhos são obtidos. No segundo os cabeçalhos são encaminhados para a etapa de *match-action*. Os *pipelines* de ingresso e egresso são utilizados para realizar essa atividade. Nos dois *pipelines* o funcionamento é semelhante,

em ambos o conteúdo dos cabeçalhos obtido é comparado com as tabelas *match-action* visando identificar uma ação que será realizada nesses cabeçalhos. No terceiro, antes do pacote sair do *switch*, o pacote é refeito com os cabeçalhos modificados. Com isso, o processo de encaminhamento é concluído [Bosshart et al. 2014b, Brum 2022].

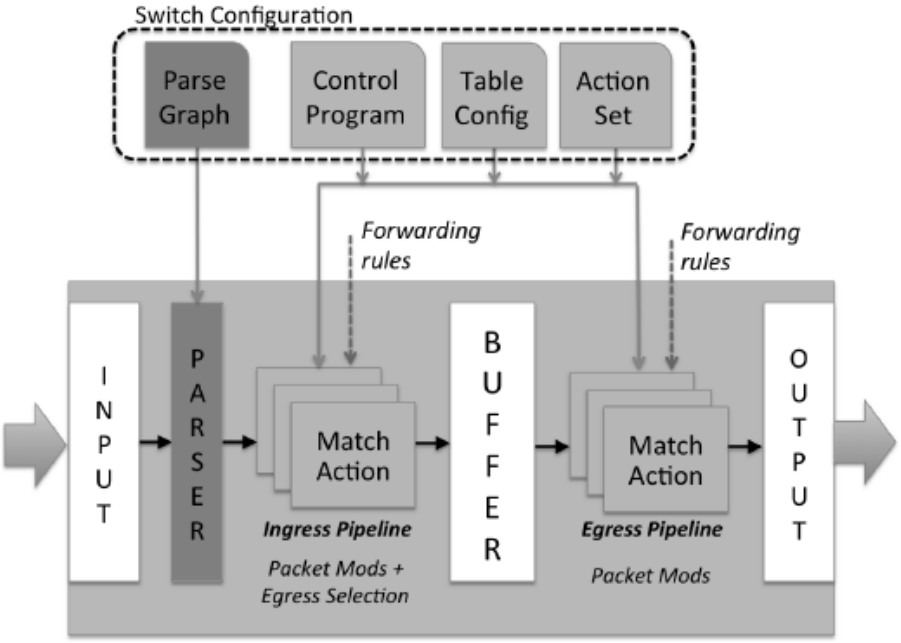

Figura 4.5: Abstração do modelo de encaminhamento PISA de dispositivos de encaminhamento programáveis (Fonte:[Bosshart et al. 2014b]).

4.4. Propriedades de Segurança de Rede

A capacidade de programar o plano de dados significa que é possível não apenas remodelar o comportamento da rede, mas também tornar a rede mais segura e confiável, melhorando sua confiabilidade, disponibilidade e integridade [Avizienis et al. 2004]. Isso pode ser feito por meio de um fluxo de serviços de segurança e confiabilidade, desenvolvidos a partir de blocos de construção provisionados diretamente nos dispositivos. Exemplos de blocos de construção incluem monitoramento e classificação de fluxo, bem como recursos de plano de dados de aplicação de políticas. Essa abordagem de provisionamento de serviços pode trazer várias vantagens exclusivas. Por exemplo, conformidade com a política pode ser garantida mesmo se o plano de controle e/ou um subconjunto de dispositivos de encaminhamento estiverem com defeito/comprometidos. Sendo assim, a medição da rede e a detecção de anomalias podem ocorrer de maneira verdadeiramente distribuída, com os *switches* acionando prontamente ações de contramedidas, se for necessário.

Além dos requisitos de desempenho, as redes modernas podem ter políticas de segurança (explícitas ou implícitas) que definem o fluxo de informação entre *hosts*. Em uma rede *multi-tenant*, por exemplo, o operador pode querer garantir que os *tenants* estejam completamente isolados uns dos outros ou que um *tenant* não possa negar ao outro acesso à rede. Várias classes de propriedades foram consideradas pela comunidade de pesquisa: independentes de contexto (propriedades agnósticas de sessões de fluxo), dependente de

contexto (referem-se aos fluxos de dados, por exemplo, iniciação da sessão), quantitativas (que são asseguradas com base em contadores, por exemplo, largura de banda garantida) e híbridas. À medida que os planos de controle e de dados se tornam mais complexos, torna-se mais difícil garantir que eles funcionem sempre corretamente. Para garantir que certas propriedades críticas sejam sempre satisfeitas, é vantajoso ter um mecanismo separado que seja apenas responsável por garantir que essas propriedades sejam respeitadas.

4.4.1. Modelagem e Análise de Políticas de Segurança

Uma maneira de expressar os requisitos que um sistema em rede deve atingir ou satisfazer é por meio de políticas de rede. A literatura é rica em soluções para especificação de políticas, verificação e aplicação. Boubata e Aib [Boutaba and Aib 2007], apresentam uma perspectiva histórica sobre a gestão de rede baseada em políticas. Os requisitos muitas vezes confiam em protocolos padrão para definir o que pode ser observado e executado (como endereços IP, portas TCP/UDP e outros campos de cabeçalho de protocolos padrão). A agenda de pesquisa de modelagem e análise de políticas para planos de dados programáveis deve se concentrar em três grandes questões: 1) como modelar e expressar políticas, 2) como traduzir/refinar políticas e 3) como lidar com conflitos entre elas.

4.4.1.1. Propriedade baseadas em políticas específicas

As soluções baseadas em políticas para o plano de dados programável deve considerar classes de propriedades para expressar requisitos de nível superior/inferior que um sistema necessita satisfazer. A questão é, quais são essas classes e propriedades? Trabalhos anteriores consideraram isolamento, acessibilidade e equivalência em SDN, mas sem fornecer uma discussão conceitual de nível superior [Khurshid et al. 2013, Lopes et al. 2015].

Uma propriedade é dita independente de contexto se for agnóstica de fluxo de sessões, ou seja, pode ser definida por pacote, sem recorrer ao estado das informações. Exemplos incluem isolamento e conectividade. Por outro lado, uma propriedade é dita dependente do contexto se aborda o fluxo de pacotes fluxos dependendo de sua semântica na rede. Um exemplo é o início da sessão, que expressa em que direção as conexões podem ser iniciadas na rede (por exemplo, um *host* pode enviar uma consulta de resolução de nomes, mas não receber um). Neste caso, diz-se que algum *host* tem permissão para iniciar uma sessão com outro. Outra classe agrega propriedades quantificáveis. Os exemplos incluem largura de banda garantida, limite de largura de banda e k-redundância. A primeira expressa uma taxa mínima que um *host* tem garantido para enviar pacotes para outro. O segundo expressa uma taxa máxima permitida para o fluxo de informações entre esses *hosts*. A terceira propriedade, k-redundância (k interpretada como uma métrica de redundância), é definida para um determinado *link* lógico e especifica a existência de k outros *links* lógicos conectando o mesmo conjunto de *hosts*. Esta propriedade pode ser útil para expressar canais de *backup* e/ou melhorar a robustez contra Ataques de negação de serviço distribuído (DDoS).

Por fim, as propriedades híbridas apresentam aquelas com características de mais de uma das classes acima. Um exemplo é o *link* equivalência, que expressa que os *links* lógicos conectando quaisquer duas entidades têm o mesmo isolamento, conectividade, lar-

gura de banda, configurações, etc. Uma noção estendida da propriedade de equivalência é a redundância k-equivalente. Um *link* é dito k-equivalente redundante se houver k outros *links* conectando o mesmo conjunto de *hosts* e com propriedades equivalentes. A oportunidade de pesquisa envolve a proposta de linguagens políticas expressivas que apoiem o nível de especificação de políticas, e que simultaneamente se aproximem mutuamente de metas conflitantes. Por exemplo, essas linguagens devem ser agnósticas do formato do cabeçalho do pacote ou da semântica de análise, mas também permitem a expressão de políticas de uma maneira que corresponda ao atual comportamento do *switch*.

4.4.1.2. Tradução de políticas de nível superior para nível inferior

Como o *hardware* de rede é personalizado sob demanda e sua semântica de análise de pacotes muda com o tempo, as soluções de especificação de políticas de segurança precisam ter uma dinâmica de revisão e atualização [Udupi et al. 2007, Craven et al. 2011]. Essas políticas em um contexto de planos de dados programáveis despertam oportunidades de pesquisas. Por exemplo: 1) Como garantir a consistência entre políticas de nível superior e inferior [Verma 2002, Westerinen et al. 2001] à medida que o comportamento do *switch* muda?; 2) Como pode-se expressar políticas baseadas em propriedades genéricas de segurança e confiabilidade, de uma forma que as torne verificáveis e aplicáveis em qualquer configuração de plano de dados? Neste contexto, é importante definir quais classes de propriedade são de interesse, bem como entender as suas implicações no projeto de mecanismos de tradução de políticas de segurança.

Em uma rede definida por *software*, cabe ao controlador garantir que as políticas de nível superior sejam mantidas [Kreutz et al. 2013]. No entanto, à medida que os aplicativos do plano de controle e os programas de comutação do plano de dados evoluem de forma independente e se tornam mais complexos, torna-se mais difícil garantir a consistência das políticas intra e internível. Esse cenário dinâmico exige soluções que vão além da tradução de políticas e também verificam inconsistências. Um exemplo é uma política declarando que duas redes A e B devem ser isoladas (um cenário de *datacenter* multilocatário) e uma permitindo pacotes do *host* $a_i \in A$ para $b_j \in B$. Outro caso é uma política que expressa que dois *hosts* estão simultaneamente isolados e conectados.

Pesquisas anteriores consideraram casos como conflitos entre diferentes tipos de políticas de nível superior [Lupu and Sloman 1999] e análise de conflito baseada em regras [Hamed and Al-Shaer 2006]. No entanto, eles são limitados, pois consideram linguagens de especificação de políticas de nível mais alto ou são fortemente acoplados a protocolos de rede tradicionais. Sendo assim, a criação de soluções que possam garantir consistência de políticas de nível superior a inferior, considerando a especificação abstrata de programas de comutação, apresenta-se como uma avenida de pesquisa promissora a ser explorada pela comunidade de pesquisa.

4.5. Verificação de Políticas de Segurança

A imposição e a verificação são abordagens complementares que podem ser aplicadas como solução para garantir que políticas de segurança sejam respeitadas. Usando a imposição, o plano de dados pode ser monitorado durante a execução para buscar e bloquear

ações que resultem em violações das políticas. A verificação (em conjunto com validação) se concentra em encontrar os *bugs* antes que os programas sejam implantados. Ela atua assegurando que o programa atenda às propriedades declaradas por seus requisitos.

Em um mundo onde os gerentes e operadores de rede podem redefinir o comportamento de dispositivos de encaminhamento, escrevendo seus próprios códigos para implementar alguma especificação de protocolo, a verificação e validação adequada (V&V) do código dos dispositivos torna-se crítica para o gerenciamento adequado das operações de rede e, portanto, a continuidade dos negócios. Em 2016, um roteador com defeito forçou a *Southwest Airlines* a cancelar 2.300 voos em quatro dias, resultando em uma perda de $ 74 milhões [Carey 2017]. Alguns anos depois (julho de 2020), uma configuração de roteador defeituosa na *Cloudflare* causou uma interrupção de rede que durou apenas 27 minutos, mas levou a uma grande interrupção dos serviços de Internet em todo o mundo por mais de uma hora [Winder 2020]. A comunidade de redes tem pesquisado soluções para lidar com defeitos de *software* antes que eles causem tais danos. Abordagens como metadados sintáticos, execução simbólica, asserções e testes funcionais têm sido aplicadas ao teste de *software* de plano de dados. Nesta seção são abordadas algumas das técnicas utilizadas para verificação e validação para *software* de plano de dados programáveis.

Tome como exemplo o trecho de código na Figura 4.6 de um programa de *switch* que faz NAT e ACL. Mesmo uma simples linha ausente como `ck.update(hdr.ipv4)` (no controle `TopDeparser`) não pode ser capturada durante o tempo de desenvolvimento sem informações extras fornecidas pelo programador (neste caso, os pacotes IPv4 de saída devem ter uma soma de verificação válida). Há casos mais complexos, no entanto. Liu et al. [Liu et al. 2018] descreveram um caso hipotético (inspirado em uma situação real de uma atualização de recurso de produto Cisco [Kazemian 2017]) em que um roteador defeituoso não estava mais aplicando as regras de ACL corretamente após uma atualização de *software*, devido a uma alteração na ordem em que o ACL e o NAT foram aplicados da versão anterior para a versão mais recente. Este caso também é mostrado na Figura 4.6, na seção `apply` do controle `TopPipe`. Outros casos, como verificar se um determinado cabeçalho é válido, podem evitar acessos para campos de cabeçalho indefinidos (como `hdr.ipv4.src_addr: lpm;` e `hdr.ipv4.dst_addr: lpm;`).

Exemplos como o da Figura 4.6 ilustram casos de defeitos de *software* por *omissão* (quando uma especificação de protocolo não está totalmente implementada no código do *switch*) e *fato incorreto* (quando o *switch* comportamento do código não está de acordo com sua especificação) [Travassos et al. 1999], respectivamente. Diversas abordagens foram desenvolvidas para verificar se um dado plano de dados satisfaz um conjunto de propriedades pretendidas [Son et al. 2013, Dobrescu and Argyraki 2014, Lopes et al. 2015, Panda et al. 2017]. No entanto, aqueles que são capazes de modelar programas P4 não podem raciocinar sobre propriedades específicas do programa. No restante desta seção, o estado da arte sobre verificação e validação de *software* de plano de encaminhamento (V&V) será revisado a partir de duas técnicas: verificação de planos de dados com asserções e via análise de fluxo de dados.


```
#include <core.p4>
#include "vss.p4"

header ethernet_t {
  bit<48> dst_addr;
  bit<48> src_addr;
  bit<16> ether_type;
}
header ipv4_t {
  bit<4> version;
  bit<4> ihl;
  bit<8> diffserv;
  bit<16> totalLen;
  bit<16> id;
  bit<3> flags;
  bit<13> fragOffset;
  bit<8> ttl;
  bit<8> protocol;
  bit<16> checksum;
  bit<32> src_addr;
  bit<32> dst_addr;
}
struct headers {
  ethernet_t eth;
  ipv4_t ipv4;
}
```

```
parser TopParser(packet_in pkt, out headers hdr) {
  state start {
    hdr.extract(pkt.eth);
    transition select(pkt.eth.ether_type) {
      0x800: parse_ipv4;
      default: accept;
    }
  }
  state parse_ipv4 {
    hdr.extract(pkt.ipv4);
    transition accept;
  }
}

control TopDeparser(inout headers hdr, packet_out pkt) {
  Ck16() ck;
  apply {
    pkt.emit(hdr.eth);
    if (hdr.ipv4.isValid()) {
      ck.clear();
      hdr.ipv4.checksum = 16w0;
      hdr.ipv4.checksum = ck.get();
    }
    pkt.emit(hdr.ipv4);
  }
}
```

```
control TopPipe(inout headers hdr, in error parseError,
    ↪ in InControl inCtrl, out OutControl outCtrl) {
  action allow() { }
  action deny() { outCtrl.outputPort = DROP_PORT; }
  action rewrite(bit<32> src_addr, PortId port) {
    hdr.ipv4.src_addr = src_addr;
    outCtrl.outputPort = port;
  }
  table acl {
    actions = { allow; deny; }
    key = { hdr.ipv4.src_addr: lpm; }
  }
  table nat {
    actions = { rewrite; }
    key = { hdr.ipv4.dst_addr: lpm; }
  }
  apply {
    if (hdr.ipv4.isValid()) {
      acl.apply();
      nat.apply();
    }
  }
}

VSS(TopParser(), TopPipe(), TopDeparser()) main;
```

Figura 4.6: Código simples NAT e ACL P4 para o modelo de switch VSS (código adaptado de [Liu et al. 2018]).

4.5.1. Verificação de Segurança usando Data Flow Analysis

Análise de fluxo de dados [Hecht 1977] tem sido amplamente utilizado para otimização de código por compiladores e detecção de anomalias de programa por meio de análise estática de programa. Em geral, classifica cada ocorrência de uma variável no programa como **definição** ou como **uso**. Essas técnicas, portanto, usam informações de fluxo de dados do programa para derivar requisitos de teste.

Para detectar o uso indevido de valores de dados devido a erros de codificação, o teste de fluxo de dados pode ser usado como critério de teste estrutural [Vergilio et al. 1997]. Rapps e Weyuker propuseram o *Def-Use Graph*, uma extensão do *Control Flow Graph* (CFG) [Rapps and Weyuker 1985]. Em sua proposta, são adicionadas informações ao CFG sobre o fluxo de dados do programa, que identifica as associações em que um valor é atribuído a uma variável (*definição de variável*) e onde esse valor é lido (*uso de variável*). Os testes de fluxo de dados são gerados a partir dessas associações. De acordo com o modelo de fluxo de dados [Vergilio et al. 1997], sempre que um valor é armazenado em um local de memória ocorre a definição da variável. É o caso quando a variável está no lado esquerdo de uma atribuição de comando, comando de entrada ou chamadas de procedimento como um parâmetro de saída.

Para gerar os testes de fluxo de dados, todos os subcaminhos são mapeados entre a atribuição de uma variável (definição) aos pontos em que a variável é utilizada (uso). Há duas maneiras de usar uma variável: computando a variável *(c-uses)*, onde um valor é usado em um cálculo ou declaração de saída; ou usando predicados *(p-uses)* que ocorrem sempre que um valor é usado em uma instrução de predicado. A notação para representar esses padrões é baseada em Rapps e Weyuker [Rapps and Weyuker 1985]:

- d – definido, inicializado
- u – usado

Existem três possibilidades para a primeira ocorrência de uma variável através de um caminho de programa. O símbolo $\sim$ é usado para denotar que antes disso a variável não existia [Rapps and Weyuker 1985]:

Tabela 4.1: Teste de anomalias (Fonte: [Rapps and Weyuker 1985]).

	Anomalia	**Descrição**
dd	Definido e definido novamente	Não inválido, mas suspeito. Possível *bug*.
du	Definido e usado	Permitido. Caso normal.
ud	Usado e definido	Permitido.
uu	Usado e usado novamente	Permitido.

1. ∼d – a variável não existe, então está definida (d)
2. ∼u – a variável não existe, então é usada (u)
3. ∼k – a variável não existe, então ela é destruída (k)

Dessas três possibilidades, apenas a primeira está correta, onde a variável não existia e então ela é definida. A segunda está incorreta porque você não pode fazer um uso seguro de uma variável a menos que ela tenha sido definida antes e a terceira provavelmente também está incorreta, porque uma variável está sendo destruída antes de ser criada. Na linguagem P4, matar ou destruir variáveis não é uma construção da linguagem.

Def-use paths (também chamado de *du-paths*) é um par ordenado (d, u), onde uma instrução chamada *d* contém uma definição de uma variável *v*, que é usada em uma instrução *u* em um programa [Rapps and Weyuker 1985]. Table 4.1 lista as combinações de uso e as consequências correspondentes.

A identificação de uma anomalia por meio do teste de fluxo de dados nem sempre representa um resultado incorreto na execução da aplicação. Embora possa ser uma anomalia inofensiva, vale a pena investigar porque muitas vezes representa um sinal de erro do programador ou más práticas de codificação.

Um método para detectar as anomalias do fluxo de dados foi desenvolvido por Fosdick e Osterweil [Fosdick and Osterweil 2011]. A ideia básica é calcular as chamadas expressões de caminho em um gráfico de fluxo usando algoritmos de análise de fluxo de dados. Uma expressão de caminho descreve todas as ações executadas em uma variável ao longo dos caminhos mapeados. Anomalias no fluxo de dados podem ser detectadas pela sequência de definições e usos que ocorrem com cada variável ao longo do caminho.

A Figura 4.7 descreve parte da análise de fluxo de dados para o código mostrado na Figura 4.6. O gráfico de fluxo de dados para este código é mostrado na parte inferior da figura, ou seja, os caminhos possíveis são mapeados e cada nó tem a anotação de uso (leitura) e definição (escrita) para cada variável. Em seguida, é realizada a análise do fluxo de dados para cada caminho. Na parte superior da figura, a análise da definição e sequência de uso para cada variável identificada para os fluxos #1 e #2. Para este código, pode-se ver um *bug* e um *bug* em potencial, de acordo com as possíveis primeiras ocorrências das variáveis e as possibilidades de pares mostradas na Tabela 4.1.

O teste de fluxo de dados é comumente aplicado para testes de unidade, ou seja, testes de uma unidade de programa. No entanto, existem investigações que estendem o teste de fluxo de dados para testes de integração [Horgan and London 1991]. Para diferentes níveis de teste (unitário, integração, sistema) são aplicados diferentes critérios de

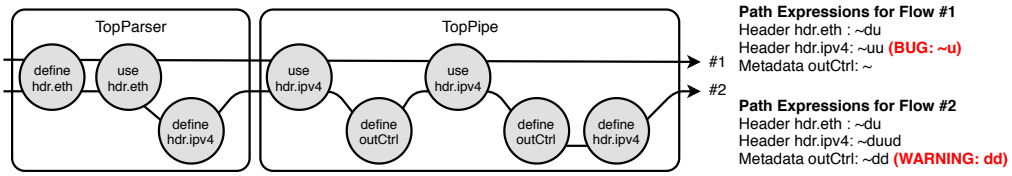

Figura 4.7: Análise parcial de fluxo de dados para o trecho de código na Figura 4.6.

teste de fluxo de dados, o que limita de alguma forma o número de caminhos explorados para identificar defeitos. Além disso, também existem trabalhos que estendem a aplicação dessa técnica de teste para testar *software* e componentes orientados a objetos.

4.5.1.1. Visão Geral

A presente seção apresenta uma visão geral de como usar a análise de fluxo de dados para descobrir problemas e vulnerabilidades de segurança em programas de *switch* P4, usando a Figura 4.8 como base. A abordagem usa uma especificação JSON do código do *switch*. Portanto, a primeira etapa do processo de verificação é a geração de uma especificação JSON a partir de um programa P4. O arquivo JSON utilizado é o mesmo esperado pelo switch BMv2, *software switch* utilizado popularmente para avaliar as funcionalidades de uma especificação de programa P4. O lado esquerdo da Figura 4.9 mostra um trecho de código basic.p4, enquanto no lado direito é mostrado o mesmo trecho de código em JSON.

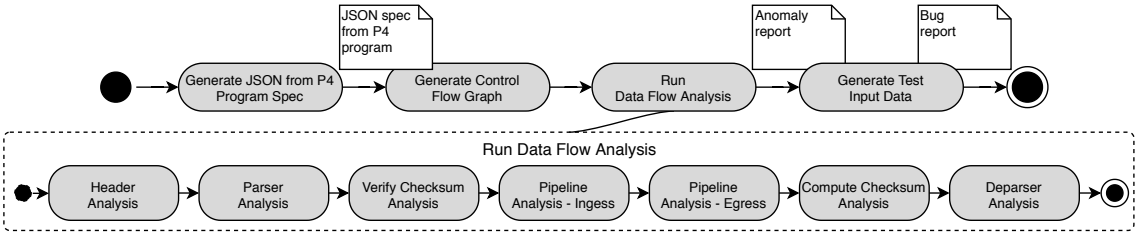

Figura 4.8: Visão geral da análise do fluxo de dados do código do switch P4.

Com base na especificação JSON, gera-se o gráfico de fluxo de controle do programa P4. Este gráfico contém todos os caminhos de execução possíveis dentro da especificação P4. Para ilustrar, o gráfico de fluxo de controle representado na Figura 4.10, do código do *switch* basic.p4[10], mostra oito caminhos de execução possíveis.

Para cada caminho possível, executa-se a análise de fluxo de dados. Este processo (detalhado na parte inferior da Figura 4.8) gera um relatório indicando expressões de caminho em cada variável e campos de cabeçalho, bem como anomalias identificadas em expressões de caminho de acordo com a teoria da análise de fluxo de dados.

[10]O código fonte do *switch* basic.p4 foi obtido dos tutoriais disponíveis na linguagem oficial do P4 7repositório do github. Uma cópia do arquivo pode ser encontrada em https://github.com/ComputerNetworks-UFRGS/p4-data-flow/blob/master/examples/basic.p4

```
/* -- P4_16 -- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;

/*************************
***** H E A D E R S *****
*************************/

typedef bit<9> egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
   macAddr_t dstAddr;
   macAddr_t srcAddr;
   bit<16> etherType;
}
```

```
{
"program" : "basic.p4",
 "_meta_" : {
  "version" : [2, 7],
  "compiler" : "https://github.com/p4lang/
      ↪ p4c"
 },
 "header_types" : [ {
   "name" : "scalars_0",
   "id" : 0,
   "fields" : []
  }, {
   "name" : "ethernet_t",
   "id" : 1,
   "fields" : [
    ["dstAddr", 48, false],
    ["srcAddr", 48, false],
    ["etherType", 16, false]
   ] },
```

Figura 4.9: Trecho de código do programa basic.p4.

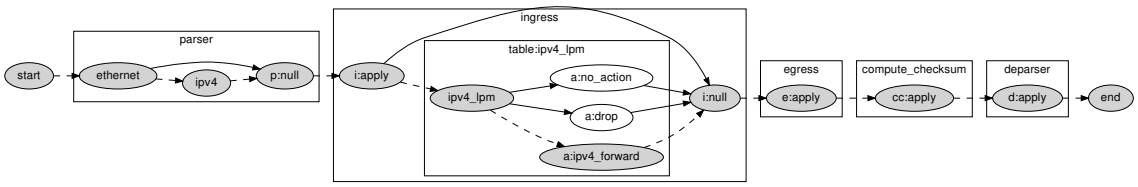

Figura 4.10: Gráfico de fluxo de controle para o programa de switch basic.p4.

A Figura 4.11 mostra as expressões de caminho obtidas para o caminho de execução destacadas nas linhas pontilhadas e elipses cinzas na Figura 4.10. Para ilustrar, considere a expressão de caminho do campo `ipv4.ttl:` DUDUU. A primeira definição (D) ocorre durante a extração do cabeçalho `packet.extract(hdr.ipv4)`. Então, um use (U) seguido por definição ocorre em `hdr.ipv4.ttl = hdr.ipv4.ttl-1`. Finalmente, dois usos ocorrem para cálculo do *checksum* e *deparsing* dos pacotes.

Existem casos de variáveis usadas mas nunca definidas, que são reportadas como **bugs**. Para outros *bugs* potenciais relatados, eles são usados como entrada para gerar pacotes de teste. Nesta etapa, implanta-se a especificação JSON no switch BMv2 e injeta-se pacotes de teste que tentam explorar as expressões de caminho anômalo. Os pacotes de teste são cuidadosamente projetados para explorar o fluxo de execução que causa a expressão do caminho anômalo e exercitá-lo. Caso ocorra um comportamento anormal do *switch* (por exemplo, um pacote que deveria ter sido descartado é encaminhado ou um pacote é descartado silenciosamente), então um *bug* é revelado.

Tendo fornecido uma visão geral da solução, a seguir apresenta-se com mais detalhes o processo de análise de fluxo de dados e a geração automatizada de pacotes de teste. Sendo assim, assume-se sem perda de generalidade o uso do modelo *V1Switch*, que é composto por blocos *parser*, *checksum*, *ingress*, *egress*, *compute checksum* e *deparser*. Observe que com esta mesma metodologia, é possível implementar uma solução funcional para outras arquiteturas existentes.

```
Execution Path:
   0 -> parsers/start -> parsers/parse_ipv4 -> parsers/null -> ingress/node_2 ->
       ↪ ingress/MyIngress.ipv4_lpm -> ingress/MyIngress.ipv4_lpm/MyIngress.
       ↪ ipv4_forward -> ingress/null -> egress/null -> compute_checksum -> deparsers
DF_Table:
   standard_metadata
      egress_spec: D
   ethernet
      dstAddr: DUDU
      srcAddr: DDU
      etherType: DUU
   ipv4
      version: DUU
      ihl: DUU
      diffserv: DUU
      totalLen: DUU
      identification: DUU
      flags: DUU
      fragOffset: DUU
      ttl: DUDUU
      protocol: DUU
      hdrChecksum: DDU
      srcAddr: DUU
      dstAddr: DUUU
      $valid$: UU
   MyIngress.ipv4_forward
      dstAddr: PU
      port: PU
```

Figura 4.11: Exemplo de expressões de caminho para o código do switch basic.p4. P significa passagem de parâmetro. O caminho segue o JSON gerado pelo compilador p4c para o modelo de switch simples bmv2.

A análise de cada caminho de execução dentro de um código de *switch* P4 compreende o processamento de cada um dos componentes do *switch* nesse caminho: definições de cabeçalho, *parser* e controles, conforme mostrado nas tarefas descritas na parte inferior da Figura 4.8. O processo realizado em cada tarefa é descrito em mais detalhes na literatura [Birnfeld et al. 2020].

A saída da etapa de análise de fluxo de dados é uma lista de possíveis *bugs*. *Bug* potencial significa que ele pode realmente ser classificado como um *bug* caso de teste possa levar a uma situação defeituosa. Assim, o último passo da proposta é verificar esses possíveis *bugs* para confirmar se realmente são *bugs* no programa.

Para exercitar esses casos, faz-se necessária uma abordagem adicional, como por exemplo um gerador de pacotes para exercitar os caminhos com potenciais *bugs*. Com isso, pode-se gerar pacotes para confirmar os possíveis *bugs* identificados em um código de *switch* executando no BMv2. Sendo assim, faz-se necessário preencher as tabelas do *switch* com um conjunto mínimo de regras que exercitam o caminho a ser explorado, e então enviar um conjunto de pacotes de teste, fixando em zero/indefinido os casos suspeitos. Caso o pacote não seja processado de acordo com o resultado esperado (*e.g.*, encaminhado quando deveria ter sido descartado), então o *bug* é confirmado.

Apenas um pequeno subconjunto de *bugs* relatados precisa dessa inspeção mais detalhada usando essa técnica de teste estrutural, como por exemplo o caso de variáveis que são definidas duas vezes sem uso no meio (*i.e.*, sua expressão de caminho tem uma

subsequência DD). A lógica é que a ocorrência de uma variável escrita duas vezes pode indicar um problema na lógica de programação do *switch*. Entre as etapas de um processo de análise de fluxo de dados de código de *switch* P4, a geração de pacotes de teste é a única que pode exigir esforço manual para gerar pacotes usando um gerador de pacotes.

4.5.1.2. Estudo de Caso de Vulnerabilidades: NAT

A aplicação de análise de fluxo de dados permitiu identificar um pequeno problema de segurança no programa simple_nat, que implementa um NAT com suporte a IPv4. A técnica revelou três *bugs* em seu código. A Figura 4.12 mostra os caminhos de execução (e respectivas expressões de caminho dos campos de cabeçalho) relacionados a dois desses *bugs*. A primeira refere-se à possibilidade de pacotes sem cabeçalho IPv4 serem processados pela tabela ipv4_lpm. Observe no primeiro caminho de execução que após analisar o cabeçalho *Ethernet (parsers/ethernet)*, o analisador vai para o estado de saída *(parsers/null)*. Este é o caso do caminho `default: accept;` no `transition select(hdr.ethernet.etherType)`, como pode ser visto no código simple_nat-16.p4 disponível no repositório. Mais tarde no caminho de execução (ingress/node_4 -> ingress/ipv4_lpm), uma condicional com erros permite que o pacote sem um cabeçalho IPv4 válido seja processado pela tabela ipv4_lpm.

O código defeituoso neste caso é `if (meta.meta.do_forward == 1w1 && hdr.ipv4.ttl > 8w0)`. Da especificação P4_16[11], campos de cabeçalho que não são definidos antes do uso podem ter um valor indefinido e, portanto, levar a mudança a um comportamento anormal. De fato, após exercitar o *bug* usando PTF, descobriu-se que um pacote sem cabeçalho IPv4 que deveria ter sido descartado na verdade foi encaminhado pelo *switch* (defeito de *software* devido a fato incorreto). Para evitar isso, os autores do código do *switch* simple_nat deveriam ter verificado a validade do cabeçalho IPv4, usando o método `hdr.ipv4.isValid()`.

O segundo *bug* está relacionado aos campos de cabeçalho TCP não extraídos, mas mesmo assim traduzidos. Observe no segundo caminho na Figura 4.12 que o cabeçalho IPv4 é analisado, mas não o cabeçalho TCP (`parsers/parse_ethernet -> parsers/parse_ipv4 -> parsers/parse_null`). Então, ao chegar em `egress/send_frame`, a ação `do_rewrites` é acionada sem verificar se o cabeçalho TCP é válido (um defeito de *software* por omissão). Por fim, o terceiro *bug* encontrado está relacionado ao uso do campo IPv4 TTL sem uma definição prévia (fato incorreto), causando uma aplicação incorreta da tabela `ipv4_lpm`.

4.5.2. Verificação de Segurança em Programas P4 com Asserções

Asserções têm sido popularmente usadas como uma abordagem de verificação de rede capaz de modelar e verificar (na compilação) propriedades gerais de segurança e correção de programas P4. Uma das soluções nesta linha é o ASSERT-P4 [Neves et al. 2018], que fornece uma linguagem de asserção expressiva permitindo aos programadores especificar suas propriedades pretendidas simplesmente anotando seus programas P4.

[11] P4_16 Specification: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

```
0 -> parsers/start -> parsers/parse_cpu_header -> parsers/parse_ethernet -> parsers/
    ↪ null -> ingress/if_info -> ingress/if_info/set_if_info -> ingress/nat ->
    ↪ ingress/nat/nat_hit_ext_to_int -> ingress/node_4 -> [continua]
[continua] -> ingress/ipv4_lpm -> ingress/ipv4_lpm/set_nhop -> ingress/forward ->
    ↪ ingress/forward/set_dmac -> ingress/null -> egress/node_9 -> egress/send_to_cpu
    ↪  -> egress/send_to_cpu/do_cpu_encap -> egress/null -> compute_checksum ->
    ↪ deparsers
   ethernet
      dstAddr: DDU
   ipv4
      version: UU
      ihl: UU
      diffserv: UU
      totalLen: UU
      identification: UU
      flags: UU
      fragOffset: UU
      ttl: UUDUU
      protocol: UUU
      srcAddr: UUU
      dstAddr: UUU
   tcp
      srcPort: UU
      dstPort: UU
      seqNo: UU
      ackNo: UU
      dataOffset: UU
      res: UU
      flags: UU
      window: UU
      urgentPtr: UU

0 -> parsers/start -> parsers/parse_cpu_header -> parsers/parse_ethernet -> parsers/
    ↪ parse_ipv4 -> parsers/null -> ingress/if_info -> ingress/if_info/set_if_info ->
    ↪  ingress/nat -> ingress/nat/nat_hit_int_to_ext -> ingress/node_4 -> [continua]
[continua] -> ingress/ipv4_lpm -> ingress/ipv4_lpm/set_nhop -> ingress/forward ->
    ↪ ingress/forward/set_dmac -> ingress/null -> egress/node_9 -> egress/send_frame
    ↪ -> egress/send_frame/do_rewrites -> egress/null -> compute_checksum ->
    ↪ deparsers
   ethernet
      dstAddr: DDU
   ipv4
      hdrChecksum: DDU
   tcp
      seqNo: UU
      ackNo: UU
      dataOffset: UU
      res: UU
      flags: UU
      window: UU
      urgentPtr: UU
```

Figura 4.12: Execution paths with faulty behavior for simple_nat.

Uma vez anotado, um programa é executado simbolicamente, com asserções sendo verificadas enquanto todos os seus caminhos são percorridos. Dado que o tempo gasto para realizar a execução simbólica cresce exponencialmente com a complexidade do programa, também apresenta-se uma variedade de técnicas de otimização que podem ser empregadas para reduzir o tempo de verificação e o número de instruções executadas. As técnicas consistem em usar paralelização da execução simbólica, sinalizadores de otimização do compilador, anotações de código para restringir os pacotes e o fluxo de controle, uma estratégia de relatório de erros para otimizar as operações de E/S e descoberta de violação de asserção e fatiar o programa para reduzir a complexidade do modelo.

Os desenvolvedores usam asserções para expressar propriedades de programas P4. Uma *linguagem de asserções* é necessária para capturar comportamentos de processamento de pacotes e facilitar a tarefa de especificar propriedades de rede complexas. Isso inclui o raciocínio sobre a formação de pacotes, encaminhamento e propriedades de fluxo de controle, cujo comportamento pode depender não apenas do estado das variáveis do programa em um local específico, mas também de como o programa manipula os pacotes em outros pontos do código. Para atingir esse objetivo, o ASSERT-P4 introduz uma linguagem de asserção usando o mecanismo de anotação de código disponível em P4. A solução define uma anotação chamada `assert`, permitindo que o desenvolvedor e/ou terceiros expressem/interpretem propriedades de forma intuitiva.

4.5.3. Visão Geral

A Figura 4.13 resume a gramática da linguagem de asserção. Assemelha-se a asserções do estilo C encontradas em linguagens de programação tradicionais, diminuindo a barreira para a adoção. No entanto, o conceito de asserção no escopo de programabilidade do plano de dados geralmente é mais amplo e, inclui elementos como *location-restricted* e *location-unrestricted*. Um elemento com restrição de localização é aquele que testa o valor de uma variável de programa onde a asserção é especificada, como em linguagens de programação tradicionais como C ou Java. Os de localização irrestrita, em contraste, aplicam-se a todo o programa de plano de dados. Eles podem ser usados, por exemplo, para garantir propriedades de nível superior que se espera que o programa satisfaça, como isolamento – afirmar que certos pacotes nunca seriam encaminhados para certas portas, ou para garantir que algumas ações sejam tomadas em certos cabeçalhos.

Sintaticamente, cada assertiva é composta por uma expressão *booleana*, que pode incluir métodos primitivos. Os métodos permitidos são `forward`, `traverse_path`, `constant`, `if`, `extract_header` e `emit_header`. Tanto as expressões quanto os métodos podem operar sobre um ou mais valores, campos de cabeçalho ou cabeçalhos. Não há diferença de sintaxe entre elementos com restrição de localização e sem restrição de localização. Semanticamente, cada asserção representa um *booleano* que deve ser avaliado como verdadeiro ou falso, onde valores e campos de cabeçalho são avaliados como verdadeiro se forem diferentes de zero e falsos caso contrário. As expressões podem ser inteiras ou *booleanas* e, em ambos os casos, com a mesma semântica que suas contrapartes na linguagem P4.

Os métodos funcionam da seguinte forma: `if(`b_1`,` b_2`, [`b_3`])` é semelhante às declarações condicionais tradicionais: se a expressão b_1 for verdadeira, então a expressão

```
b ::= v                 m ::= forward()
    | f                     | traverse_path()
    | m                     | constant(f)
    | !b                    | if(b, b, [b])
    | b || b                | extract_header(h)
    | b && b                | emit_header(h)
    | b == b            i ::= v
    | b != b                | f
    | i >= i                | i * i
    | i <= i                | i / i
    | i < i                 | i % i
    | i > i                 | i + i
    | i == i                | i - i
    | i != i
```

Figura 4.13: Gramática da linguagem de asserção.

b_2 será avaliada, caso contrário a alternativa b_3 será avaliada. Este é o único método com restrição de localização, com todos os outros sendo irrestritos. `traverse_path()` indica se uma determinada estrutura no programa (por exemplo, uma ação) será percorrida antes que a execução do programa termine. `constant(f)` é verdadeiro se o campo `f` não mudar a partir do local de asserção, ou seja, até que o programa termine. `forward()` retorna *true* quando o pacote não for descartado no final do programa. `extract_header(h)` é verdadeiro se um cabeçalho `h` foi ou será extraído do pacote. Por fim, `emit_header(h)` retorna *true* se o pacote for transmitido com o cabeçalho `h`.

Os métodos apresentados na linguagem permitem a especificação de tipos de propriedades que seriam difíceis ou impossíveis de expressar usando apenas asserções tradicionais. A adição de `forward()` permite a expressão de propriedades de encaminhamento, que são essenciais para programas de plano de dados. `traverse_path()` permite raciocinar sobre o fluxo de controle do código fonte. `constant()` facilita a verificação da integridade das variáveis em todo o programa. Tanto `extract_header()` quanto `emit_header()` permitem a expressão das propriedades de formação de pacotes no nível do analisador e do analisador, respectivamente. Finalmente, `if()` auxilia o processo de combinação de métodos e expressões em uma expressão condicional.

A Figura 4.14 mostra um exemplo de um programa P4 anotado, com asserções em negrito. Por clareza, apenas as partes mais relevantes do programa são exibidas. Este programa descreve um *pipeline* de processamento de pacotes com uma única tabela (`dmac`), que é instanciada dentro do bloco de controle `TopPipe`. Cada entrada da tabela pode invocar uma das duas ações (`Drop()` ou `Set_dmac()`). As assertivas visam verificar se: (i) pacotes marcados para *drop* nunca são encaminhados (linha 7), e (ii) somente pacotes com TTL maior que zero são encaminhados (linha 21). As duas asserções contêm elementos sem restrição de localização (por exemplo, `forward()` captura o estado do programa no final de sua execução) e elementos com restrição de localização (por exemplo, a expressão `headers.ip.ttl >0` testa o valor de `headers.ip.ttl` no ponto em que a asserção é encontrada).


```
1  ...
2 control TopPipe(inout Parsed_packet headers,
3                   out OutControl outCtrl) {
4  ...
5  action Drop() {
6    outCtrl.outputPort = DROP_PORT;
7    @assert("if(traverse_path(), !forward())");
8 }
9  action Set_dmac(EthernetAddress dmac) {
10   headers.ethernet.dstAddr = dmac;
11 }
12 table dmac {
13   key = { nextHop : exact; }
14   actions = { Drop; Set_dmac; }
15   default_action = Drop;
16 }
17 apply {
18   ...
19   dmac.apply();
20   ...
21   @assert("if(forward(), headers.ip.ttl > 0)");
22 }
23}
```

Figura 4.14: Exemplo de um programa P4 anotado.

4.5.4. Estudo de Caso: Verificação de Programas P4

Primeiro, demonstra-se a eficácia da proposta para encontrar *bugs* e violações de políticas em planos de dados programáveis. Usa-se asserções para encontrar *bugs* em quatro aplicativos P4 recentes e confirma-se examinando manualmente os códigos-fonte.

Dapper [Ghasemi et al. 2017]: Dapper é uma ferramenta de diagnóstico de desempenho de plano de dados que infere gargalos TCP analisando pacotes em tempo real. Coloca-se um conjunto de asserções básicas no início do bloco de controle de ingresso, e através da asserção `if(ipv4.ttl == 0, !forward())`, descobre-se que o Dapper pode encaminhar pacotes IPv4 mesmo quando o campo TTL é zero. Por inspeção manual, nota-se que mesmo que o campo TTL seja decrementado conforme o esperado, seu valor nunca é verificado antes do encaminhamento. Por causa desse *bug*, ao depurar uma rede com um *loop*, os dispositivos baseados em Dapper podem continuar encaminhando pacotes para sempre. O protótipo encontrou esse *bug* em menos de um segundo.

NetPaxos [Dang et al. 2016]: NetPaxos é uma implementação baseada em rede do protocolo de consenso Paxos. Existem dois tipos diferentes de programas P4 nesta aplicação, um para Líderes/Coordenadores e outro para Aceitadores. Todos os outros atores são considerados totalmente implementados em *hosts* finais. Examina-se a versão atual da implementação e o conjunto de regras de encaminhamento disponibilizado pelos autores, acrescentando assertivas ao seu código. Como parte da implementação, um *Acceptor* vota adicionando informações de votação aos pacotes recebidos antes de encaminhá-los. Especificamente, a execução violou a asserção `if(traverse_path(), forward())`, localizada dentro da ação que realiza o voto. Isso indica que há pacotes

válidos (contendo informações de votação) sendo descartados. Ao inspecionar manualmente o código, descobre-se que o problema ocorre porque os pacotes são marcados primeiro para serem descartados por outra ação e não desmarcados pelas ações de votação. Esse *bug* pode ser corrigido marcando os pacotes a serem encaminhados dentro das ações que realizam a votação. De acordo com o *feedback* dos autores sobre esse *bug*, o código foi portado para $P4_{16}$, deixando a base de código antiga sem manutenção e exposta a *bugs*. O protótipo encontrou essa violação de afirmação em menos de um segundo.

DC.p4 [Sivaraman et al. 2015]: DC.p4 implementa o comportamento de um *switch* de *data center*. Ele contém várias funcionalidades, como encaminhamento L2/L3, ECMP, VLAN, espelhamento de pacotes, encapsulamento e várias ACLs (ou seja, L2, L3 ou com base em cabeçalhos mais específicos). Este programa contém mais de 2500 linhas de código distribuídas em 37 tabelas. As tabelas, por sua vez, são organizadas assumindo dois *pipelines* sequenciais de processamento de pacotes, um para pacotes de entrada/entrada e outro para pacotes de saída/saída, intercalados por um sistema de filas.

A configuração da tabela L3 ACL é verificada para descartar o tráfego com um endereço IP de destino específico para filtrar adequadamente esse tipo de pacote. Usa-se a asserção `if(ipv4.dstAddr == FILTER_ADDR, !forward())` para expressar que pacotes com endereços de destino IPv4 iguais a `FILTER_ADDR` devem ser descartados. Descobre-se que apenas configurar a L3 ACL não é suficiente para descartar pacotes IPv4, independentemente da política que está sendo aplicada. De fato, verifica-se que a L3 ACL apenas sinaliza pacotes para serem filtrados por outro módulo do sistema, que também deve ser configurado adequadamente. Embora este não seja um *bug* real, ainda é uma configuração incorreta no programa.

Switch [P4.org 2018]: Desde a introdução do documento DC.p4, sua base de código evoluiu para o programa Switch.p4, que é mantido ativamente. Usa-se abordagem de verificação para reproduzir dois *bugs* conhecidos, relatados em seu repositório. A primeira é a modificação de um campo de um cabeçalho inválido.[12] O *bug* é replicado testando com uma afirmação se o cabeçalho é válido antes de definir seu Campos. O segundo *bug* está relacionado ao encapsulamento de túnel[13], onde os cabeçalhos encapsulados são substituídos sempre que vários níveis aninhados estão presentes. Incluí-se uma declaração para testar se os cabeçalhos internos não são válidos antes de realizar o encapsulamento. A asserção falhou, confirmando que os cabeçalhos encapsulados podem ser substituídos e seu conteúdo original, descartado.

4.6. Imposição (*Enforcement*) de Políticas de Segurança

Uma alternativa à verificação é a imposição (enforcement). Em vez de verificar se uma configuração de rede está correta, um *kernel* de segurança logicamente separado evita ações que violem a política de segurança. O *kernel* de segurança deve mediar todas as ações de manipulação de pacotes no plano de dados. Ao contrário do modo de verificação, onde verifica-se as violações da política antes de uma configuração ser enviada para a rede, no modo de imposição, verifica-se as violações da política, uma vez que estão prestes a ocorrer. Tanto a verificação como a imposição têm suas vantagens e desvantagens.

[12] https://github.com/p4lang/switch/pull/102

[13] https://github.com/p4lang/switch/issues/97

Por um lado, a verificação capta problemas precocemente; um verificador pode fornecer informações de diagnóstico detalhadas sobre por que uma configuração viola uma política durante a fase de verificação. No regime de imposição, os problemas são detectados à medida que ocorrem. A imposição pode ser mais atrativa do que a verificação, porque não depende da complexidade do programa de controle ou do plano de dados.

Sendo assim, emergem benefícios para imposição (*enforcement* da política de segurança em plano de dados programáveis. Os planos permitem que os operadores de rede modifiquem o *pipeline* de processamento de pacotes dos dispositivos de rede para implementar novos protocolos, personalizar o comportamento da rede e estabelecer serviços de rede avançados. No que pese a sua simplicidade da programação, os programas P4 demonstraram ser propensos a uma variedade de *bugs* e erros de configuração [Stoenescu et al. 2016, Freire et al. 2018]. Como resultado, os operadores de rede precisam de estruturas para garantir que os programas que produzem tenham um comportamento correto para obter os benefícios de um ecossistema de *software* de plano de dados. Ferramentas de verificação de rede de última geração podem obter um modelo da rede, sua configuração e um conjunto de propriedades específicas usando formalismos tradicionais (por exemplo, lógica temporal ou regras de *Datalog*) e verificar automaticamente se essas propriedades são válidas para qualquer pacote [Beckett et al. 2017, Lopes et al. 2015].

Embora essas ferramentas ajudem os operadores de rede a identificar *bugs* antes que eles se manifestem, deve-se considerar: (i) Primeiro que a maioria dessas ferramentas exige que os programadores modelem manualmente os planos de dados programáveis, atividade complexa e propensa a erros [Lopes et al. 2015]; (ii) Em segundo lugar, essas ferramentas são geralmente restritas em termos de propriedades de acessibilidade para reduzir os tempos de verificação [Lopes et al. 2016]; (iii) Terceiro, ferramentas mais expressivas capazes de verificar múltiplas propriedades frequentemente enfrentam problemas graves de escalabilidade (por exemplo, verificar a conformidade com uma especificação de protocolo pode levar dias, mesmo para um único plano de dados programáveis; e (iv) Por fim, os programadores precisam ter habilidades técnicas formais de verificação para especificar corretamente suas propriedades.

Neves et al. [Neves et al. 2021] apresentam uma nova abordagem baseada na aplicação dinâmica (ou em tempo de execução) em vez de verificação estática. Essa abordagem tem várias vantagens práticas. Já que não é necessário esperar pelo resultado de um longo processo de verificação para enviar uma nova configuração para os *switches* de rede. Sendo assim, a aplicação do tempo de execução pode intervir prontamente se situações problemáticas realmente ocorrerem, possibilitando: obter informações úteis do código com *bugs* quando ele tem um comportamento correto e reparar problemas sem interferir em qualquer serviço de rede.

Em contraposição com a verificação estática, a aplicação do tempo de execução também permite ao programador expressar a política e o mecanismo usando o mesmo ambiente de programação que o resto do programa. Esse valor deve ser considerado, não só porque facilita a vida do programador, como evita também erros de tradução entre a implementação e as políticas. Sendo assim, para perceber os benefícios de uma aplicação dinâmica, Neves et al. [Neves et al. 2021] desenvolveram o P4box, um sistema para implantação de monitores de tempo de execução em planos de dados programáveis.

Usando P4box os programadores podem anexar monitores antes e depois dos blocos de controle, transições de estado do analisador e chamadas para funções externas de um programa P4. Cada monitor pode modificar a entrada e saída do bloco de código ou função que monitora, permitindo a verificação de pré e pós-condições a serem utilizadas para impor propriedades específicas ou modificar o comportamento do bloco monitorado.

Um monitor de tempo de execução insere-se na interação de um bloco de controle P4 ou analisador com o restante do ambiente de execução, como apresentado na Figura 4.15, permitindo que o programador do monitor modifique o comportamento do bloco P4 incluso com o restante do ambiente. Um bloco programável P4 faz a *interface* com o restante do ambiente de execução P4 na entrada no bloco, retornar do bloco as chamadas para funções externas fornecidas pela arquitetura. Na programação do modelo P4box, quando um bloco programável é invocado, o controle passa primeiro para um monitor, também escrito em P4, antes de passar para o bloco programável pretendido. Da mesma forma, quando um bloco programável completa o processamento, o controle passa primeiro para o monitor antes de retornar ao dispositivo, permitindo que um monitor modifique o comportamento de blocos programáveis de maneira bem definida.

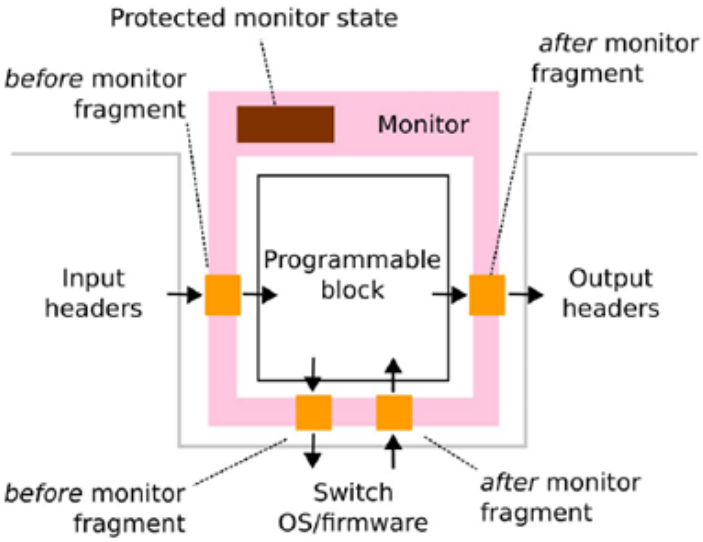

Figura 4.15: Modelo de programação P4box.

4.6.1. Monitores de Segurança para Switches P4

Um monitor de programa é uma construção de linguagem que foi desenvolvida (como uma extensão para P4) inspirada no paradigma *Aspect-Oriented Programming* (AOP) (Aspect-Oriented Programming) [Kiczales et al. 1997], o qual fornece construções em nível de linguagem para anexar código a pontos designados em um programa existente sem modificar o programa em si. Os programadores podem usar monitores para modificar ou verificar o comportamento de blocos de controle, analisadores e funções externas de programas P4 e, assim, garantir que eles respeitem um conjunto de propriedades desejadas. Os monitores são particularmente adequados ao contexto em que os programas de plano de dados são montados a partir dos módulos mantidos externamente, onde pode ser desejável alterar ou verificar o comportamento desses módulos sem modificar seu código.

Os monitores também podem interpor as chamadas para funções externas: quando um bloco programável invoca uma função externa, o controle passa primeiro para o monitor, depois para a função e depois volta para o monitor novamente, antes de retornar ao bloco programável. Um monitor pode, assim, modificar o comportamento aparente de uma função externa. Os monitores são declarados e definidos no nível superior de um programa P4, juntamente com os blocos de controle, blocos do analisador e outras declarações de nível superior. A Figura 4.16 apresenta a sintaxe para um monitor.

```
monitor <name> ( [param-list] ) on <object> {
    [local-declarations]
    (before | after) { <p4-statements> }
}
```

Figura 4.16: Sintaxe para um monitor.

Cada monitor é identificado por um único <nome> e pode receber parâmetros adicionais (<param-list>) contendo cabeçalhos e metadados, além dos parâmetros do objeto monitorado. Cada monitor deve estar associado a um plano de dados <object>, que pode ser um analisador, bloco de controle ou função externa. O tipo de recurso define o conjunto de elementos <p4-statements> que o monitor suporta. Os monitores podem ter dois tipos de métodos, a saber: antes e depois, que especificam o código fragmentos que são executados antes e depois do recurso monitorado, respectivamente. Finalmente, eles também podem conter declarações locais (por exemplo, ações, tabelas) visíveis dentro do monitor, mas não no bloco monitorado.

O P4box instrumenta um programa P4 com monitores em tempo de compilação de tal forma que o primeiro não pode contornar ou interferir neste último. O programa P4 original e os monitores de tempo de execução de definição de fonte P4 são fornecidos ao P4box, que combina o programa original com os monitores no nível intermediário para produzir um novo programa adequado para posterior compilação. No final, o código em nível de máquina contendo todos os monitores é gerado para uma variedade de destinos. Durante o processo de instrumentação, o P4box aproveita os recursos de linguagem fornecidos pelo P4, como escopos e *namespaces* separados, além da análise estática, para fornecer as seguintes garantias para cada monitor: 1) Mediação completa: o fluxo de execução do programa de plano de dados original sempre passará por um monitor (quando for definido pelo programador). Isso significa que não é possível para o programa original contornar um monitor; e 2) Não interferência: o programa original não pode interferir na operação de um monitor modificando suas variáveis locais ou cabeçalhos, o que significa que os monitores são completamente isolados do programa do plano de dados.

Juntas, as propriedades de mediação completa e não interferência permitem que os monitores restrinjam o que o programa P4 original permite realizar mesmo quando o último não é confiável (por exemplo, um programa de terceiros). Os monitores são, portanto, não apenas um mecanismo de estruturação de programa P4 orientado a aspectos, mas também uma caixa de proteção de *software* que pode ser usada para encapsular código P4 não confiável ou com erros. A Figura 4.17 mostra o fluxo de trabalho do P4box.

Os monitores podem ser combinados para impor propriedade mais complexas,

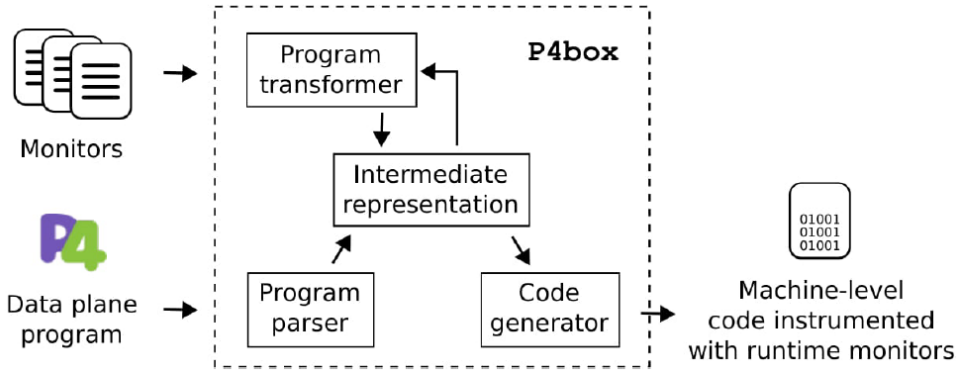

Figura 4.17: Fluxo de trabalho do P4box.

como as que envolvem extração e emissão de rótulos em pacotes. O P4box pode impor as propriedades de *(waypointing)* verificando e atualizando os rótulos sempre que esses pacotes cruzarem um dispositivo na cadeia. Como exemplo, a Figura 4.18 mostra um cenário em que os pacotes vindos de uma rede externa (ou seja, através de roteador R) deve primeiro ser inspecionado por um sistema IDS antes de chegar a um servidor web (*hosts* H1-H3). Neste caso, um monitor P4box em R introduz rótulos em cada pacote para impor o *(waypointing)*. Esses rótulos são então atualizados por outro monitor no *switch* S1 e um terceiro monitor os verifica no *switch* S2 para descartar pacotes destinados aos servidores web e não contém a *tag* atualizada (L1).

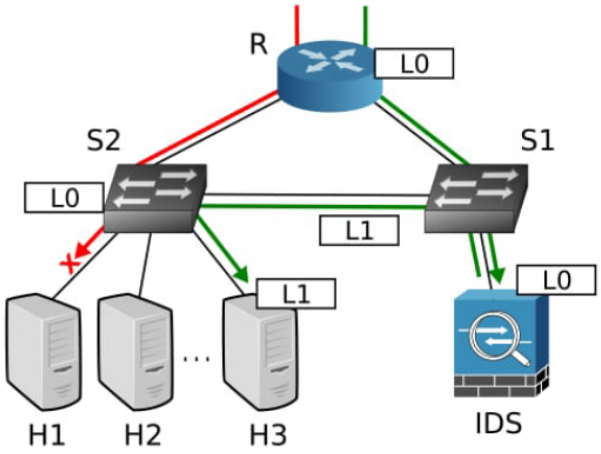

Figura 4.18: Exemplo de topologia para o *waypointing*.

A Figura 4.19 mostra como o P4box interage com o programa P4 para impor o *(waypointing)*, onde as setas verticais representam o fluxo de execução. Observe que o P4box prende o programa em três pontos: (i) primeiro: entre o análise dos cabeçalhos *Ethernet* e IPv4, para verificar se o pacote contém um rótulo e extrair o último; (ii) segundo: logo antes do início do pipeline de ação de correspondência, para operar no rótulo (por exemplo, verificar, atualizar ou remover) dependendo de como o dispositivo está conectado na topologia; e (iii) terceiro: para emitir o rótulo durante a fase de análise.

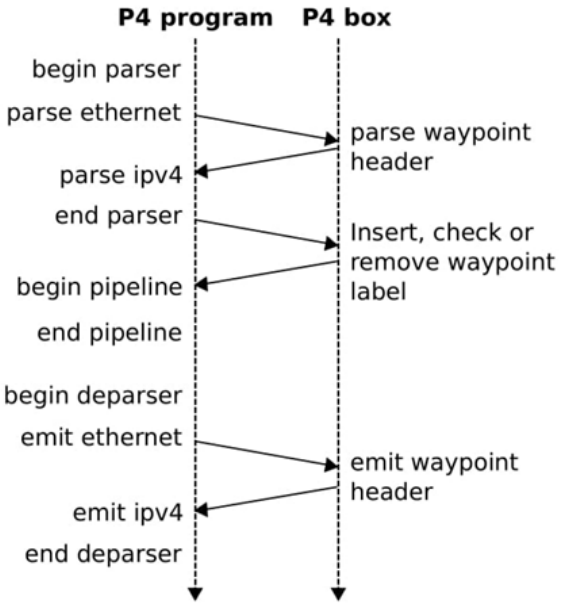

Figura 4.19: Interação entre o P4box e o programa P4 para aplicar o *waypointing*.

O P4Box suporta três tipos de monitores, a saber: 1) Monitores de bloco de controle: são responsáveis por garantir que um cabeçalho não seja modificado erroneamente pelo programa de plano de dados. O monitor é anexado ao *pipeline* de processamento e possui dois elementos: (i) antes do bloco programável, ele obtêm o estado do pacote original assim que é analisado (l.5-8); e (ii) após a bloco, ele testa se os cabeçalhos monitorados foram modificados (l.10-17). A Figura 4.20 mostra um exemplo de monitor de bloco de controle, que pode ser usado para detectar e processar informações sobrescrevendo *bugs*; 2) Monitor *parser*: podem ser anexados aos analisadores de nível superior. Como tal, antes e depois podem conter máquinas de estado finito e ambas devem ter um estado inicial e de aceitação. É possível especializar um monitor *parser* para um estado de *parser* específico, caso em que antes e depois são associados apenas a este último. Um exemplo de monitor analisador é mostrado na Figura 4.21 nas linhas 6 a 17, onde o monitor é anexado ao estado *ethernet* analisador e usado para extrair uma imposição cabeçalho. Esses monitores são particularmente úteis para pular a extração de *bits* de pacote que por algum motivo (por exemplo, confidencialidade) não devem ser visíveis para o programa de plano de dados; e 3) Monitores externos: são anexados as chamadas externas. Seus recursos são restritos ao que as ações podem fazer em P4 devido às limitações que o último tem em relação as chamadas externas (por exemplo, não é possível fazer declarações locais ou invocar uma tabela de dentro de uma ação). Semelhante aos monitores *parser*, os monitores externos também podem ser especializados em subgrupos de um recurso. Nesse caso, uma assinatura é usada para aplicar um monitor apenas a um subconjunto das chamadas externas. Um exemplo é apresentado na Figura 4.21 nas linhas 20 a 24, onde o monitor externo é aplicado apenas para chamadas para emissão de cabeçalhos do tipo *ethernet-t*. Os monitores externos são úteis para acompanhar como o programa de plano de dados interage com a plataforma subjacente.


```
1  monitor hdrInvMonitor() on Pipeline {
2    ipv4_t protec_ipv4;
3    udp_t  protec_udp;
4
5    before {
6      protec_ipv4 = hdr.inner_ipv4;
7      protec_udp = hdr.inner_udp;
8    }
9
10   after {
11     if( protec_ipv4 != hdr.inner_ipv4 ||
12         protec_udp != hdr.inner_udp ){
13       /*Run enforcement action
14         (e.g., restore original header
15         value, notify the control plane,
16         write log) */
17   }}
15 }
```

Figura 4.20: Exemplo para o monitor bloco de controle.

4.7. Tópicos Avançados

A maioria das funções de segurança de rede baseia-se em duas abordagens básicas de perfil de tráfego de plano de dados: inspeção de pacotes e análise de fluxo. A sua utilização no plano de dados tem o potencial de servir para vários fins, apoiando novos mecanismos de detecção de intrusão e serviços de verificação de políticas de segurança. Portanto, é crítico fornecer essas duas funções no plano de dados programável.

Existem vários desafios relacionados à gerência de segurança que serão brevemente cobertos usando o conceito de Programabilidade do Plano de Dados. A flexibilidade encontrada nas infraestruturas programáveis em conjunto com interfaces abertas e bem definidas permitem o desenvolvimento de ferramentas de segurança adequadas a planos de dados programáveis.

Esta seção inicial é organizada da seguinte forma. Inicialmente é apresentada a exploração de planos de dados programáveis para detectar ataques distribuídos de negação de serviço (*Distributed Denial of Service*- DDoS). Finalmente, depuração e rastreabilidade de aplicações em planos de dados programáveis é discutida.

4.7.1. Explorando Planos de Dados Programáveis para Detectar Ataques DDoS

Ataques de negação de serviço distribuído (DDoS) fazem uso dos limites de capacidade específicos aplicados a todos os recursos da rede. Esses ataques dependem de *botnet* para esgotar recursos computacionais e interromper aplicações na Internet [Hoque et al. 2015]. Buscam encaminhar um grande número de solicitações para o recurso tecnológico invadido, visando exceder a sua capacidade, interrompendo o seu funcionamento.

À medida que os *botnets* aumentam a sua aplicabilidade para explorar os dispositivos IoT (Internet das Coisas) vulneráveis, a frequência, a capacidade e o volume dos ataques DDoS amplia o seu alcance drasticamente. A detecção dessa ameaça é o pri-


```
1  struct p4boxState {
2    waypoint_t wp_header;
3  }
4
5  //Parser monitor to extract enforcement header
6  monitor wpParser(inout p4boxState pstate) on ParserImpl {
7    after parse_ethernet {
8      state start {
9        transition select(packet.lookahead<bit<32>>()){
10         16w0xFFFF : parse_wp_header;
11         default : accept;
12       }
13     }
14     state parse_wp_header {
15       packet.extract(pstate.wp_header);
16       transition accept;
17 }}}
18
19 //Extern monitor to emit enforcement header
20 monitor wpExtern(inout p4boxState pstate)
21                            on emit<ethernet_t>{
22   after {
23     packet.emit(pstate.wp_header);
24 }}
25
26 monitor wpControl(inout p4boxState pstate) on Pipeline {
27   ...
28   table check_waypoint {...}
29   ...
30
31   before {
32     //Enforce waypointing property
33     insert_label.apply();
34     check_waypoint.apply();
35     remove_label.apply();
36 }}
```

Figura 4.21: Exemplo para o monitor *parser* e monitor externo

meiro passo para minimizar as perdas por meio do desencadeamento das medidas defensivas, no entanto, representa um desafio para a pesquisa em rede [Antonakakis et al. 2017, Anstee et al. 2017, Zargar et al. 2013].

Preferencialmente, a detecção e o bloqueio de ataques DDoS devem ocorrer nas fontes para economizar esforços de deslocamento e processamento sobre o tráfego indesejado [Gil and Poletto 2001, Mirkovic et al. 2002, Peng et al. 2004]. No entanto, isso é impedido pela disseminação da atividade maliciosa, que é construída a partir da sincronização de solicitações aparentemente legítimas. Além disso, essas fontes normalmente pertencem a diferentes domínios administrativos, nos quais as políticas de segurança são definidas de forma independente. Mais adiante, nas proximidades da vítima, apesar do tráfego de ataque ser mais proeminente para detecção [Kim et al. 2006, Hoque et al. 2015], ele pode já ter saturado recursos *in-path*. A alternativa é implantar medidas defensivas em Provedores de Serviços de Internet (ISPs), que gerenciam a comunicação [Haq et al. 2015, Kang et al. 2016]. Os ISPs se beneficiam de uma visão privilegiada do tráfego e contam com *links* de alta taxa de transferência, permitindo que eles descubram e impeçam as ameaças em tempo hábil.

Ao contrário dos *datacenters*, onde o monitoramento de rede sofisticado pode ser realizado em *hosts* finais [Moshref et al. 2016, Yu et al. 2011], os ISPs dependem de *switch primitive* como amostragem de pacotes [CiscoNetworks 2017, Sflow 2017] e contagem baseada em fluxo [McKeown et al. 2008]. Os dados resultantes são então normalmente montados em servidores fora de banda para inspeção. Enquanto essas primitivas apresentam compensações entre granularidade de visibilidade, utilização de largura de banda, espaço de memória e a comunicação com servidores externos incorre em uma latência adicional para detectar eventos de rede [Moshref et al. 2013]. A fim de manter a utilização razoável da largura de banda e a carga de processamento, a amostragem de pacotes é geralmente empregada em taxas agressivamente baixas [Phaal 2009], apenas transmitindo informações de um conjunto limitado de pacotes. Diferentemente, a contagem baseada em fluxo, como em *switches* OF [McKeown et al. 2008], fornece valores exatos para métricas volumétricas com um alto custo de entradas nas tabelas.

Como alternativa promissora para este problema, o conceito emergente de programabilidade do plano de dados oferece flexibilidade para a execução de algoritmos nos *switches* de rede [Bosshart et al. 2014b]. Assumindo um fluxo de pacotes como entrada, esses algoritmos são modelados como um *pipeline* de primitivas elementares, acessos à memória e pesquisas em tabelas. Sendo assim, os operadores podem definir funções de monitoramento e delegá-las a dispositivos de plano de dados em toda rede. Essa arquitetura pode ser explorada para realizar a inspeção em cada pacote sem incorrer em sobrecarga de comunicação. No entanto, buscando executar a taxa de linha com custos razoáveis, o processamento de pacotes é restrito a um pequeno orçamento de tempo e uma quantidade limitada de memória por estágio de *pipeline* [Bosshart et al. 2013].

Lapolli et. al [Lapolli et al. 2019] desenvolveram uma arquitetura de sistema para detecção de DDoS, na qual o plano de dados responsável pela coleta do fluxo das métricas e sua inspeção. Isso é apresentado na forma de uma detecção de ataque DDoS em banda sistema totalmente implementável em uma chave programável através de P4. O trabalho compreende um *pipeline* de processamento para estimar as entropias dos endereços IP de origem e destino. Esses valores são usados para caracterizar o tráfego supostamente legítimo em tempo real. Os resultados desta caracterização servem para calcular a detecção limiares considerando um coeficiente de sensibilidade parametrizável. A fim de respeitar o rigoroso orçamento de tempo e restrições de memória para o cálculo da entropia, a frequência de endereços IP distintos é aproximada por esboços de contagem aprimorados [Charikar et al. 2002]. Outras funções aritméticas de computação intensiva são resolvidas com a ajuda de uma tabela de pesquisa *Longest Match Routing Rule* (LPM) otimizada para memória.

O sistema de detecção de DDoS é composto por um único *switch* programável que executa de forma independente a análise estatística em banda para detectar ataques DDoS. Como esses ataques são caracterizados por uma grande quantidade de *hosts* convergindo o tráfego para uma ou poucas vítimas [Haq et al. 2015], as distribuições de endereços IP de origem e destino tendem a se desviar de seu padrão normal na presença de tráfego malicioso. A entropia de Shannon [Shannon 1948] é frequentemente utilizada como forma de identificar esse desvio, apresentando alta precisão para este objetivo [Lakhina et al. 2005, Bhuyan et al. 2015]. Com isso, a abordagem sugerida busca estimar as entropias dos endereços IP e caracterizar os endereços supostamente legítimos o

tráfego para definir limites de detecção.

A Figura 4.22 apresenta uma visão da arquitetura do sistema. Para representar os endereços de tráfego recentes de distribuição, os blocos de estimativa de entropia operam em partições consecutivas do fluxo de pacotes de entrada. Essas partições, denominadas janelas de observação, contêm um número predeterminado de pacotes de interesse de reduzir os requisitos de processamento. A estimativa é construída como um algoritmo de *streaming* a ser implementado em um *pipeline* de processamento. Ao final de cada janela de observação, os blocos de caracterização do tráfego obtêm os valores de entropia para atualizar o modelo de tráfego legítimo. Por sua vez, o bloco de detecção de anomalias calcula os limites de detecção em função deste modelo e emite um alarme de ataque quando são superados pelas últimas estimativas de entropia. Esse alarme é realimentado para a detecção da anomalia nos blocos de caracterização de tráfego para que seus modelos considerem apenas tráfego supostamente legítimo.

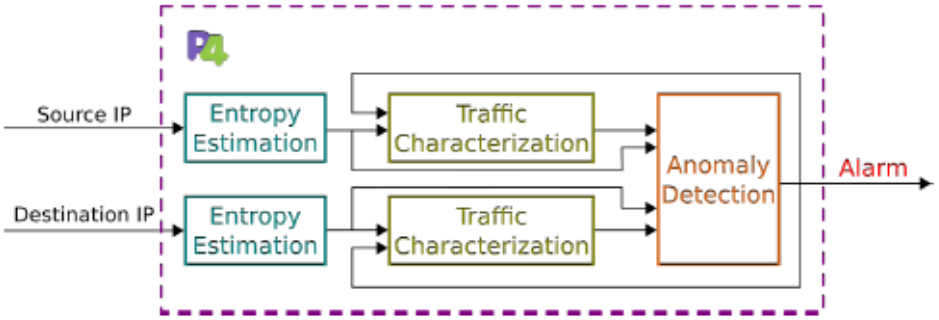

Figura 4.22: Arquitetura do sistema de detecção de ataques DDoS.

4.7.2. Depuração e Rastreabilidade de Aplicações em Planos de Dados Programáveis

Planos de dados programáveis permitem que a execução de aplicações cruzem a fronteira entre servidores x86 tradicionais e a rede de computadores, habilitando o descarregamento (ou seja, *offloading*) de partes da computação para PDPs. Esse paradigma tem sido chamado de *in-network computing* [Benson 2019]. À luz desse desenvolvimento, tanto a indústria quanto pesquisadores começaram a investigar ativamente novos projetos para aplicações distribuídas a fim de melhorar o desempenho, a escalabilidade ou a confiabilidade dessas, transferindo parte de sua funcionalidade para a rede. Dessa forma, uma vasta gama de problemas tem explorado essa possibilidade de descarregar parte da computação para a rede: *Caching*: NetCache [Jin et al. 2017] armazena em cache pares de chave-valor em switches, evitando potencialmente longos RTTs para acessar um servidor de armazenamento de chave-valor remoto; *Agregação de Dados*: DAIET [Sapio et al. 2017] realiza agregação de dados na rede para maior escalabilidade; *Machine Learning*: machine learning dentro de switches pode mitigar gargalos existentes durante o treinamento distribuído de modelos [Sanvito et al. 2018, Xiong and Zilberman 2019]; *Pattern Matching*: a correspondência de padrões eficiente pode ser alcançada através da realização de parte da computação na rede [Jepsen et al. 2019].

À medida que essas abordagens recém-descobertas se aproximam da implanta-

ção, surgem preocupações práticas sobre seu gerenciamento em tempo de execução, porque as aplicações distribuídas agora podem executar parcialmente no plano de dados. Especificamente, a incorporação de lógica em PDPs adicionou outra camada de complexidade para rastrear e solucionar problemas dessas aplicações, e esforços tradicionais de rastreabilidade e observabilidade de aplicações em servidores x86 tradicionais não se traduzem diretamente para *in-network computing* [Benson 2019]. Em particular, switches programáveis atuais não fornecem uma abstração rica o suficiente para suportar técnicas de rastreamento tradicionais [Sigelman et al. 2010, Mace and Fonseca 2018, Chow et al. 2014], e essa falta de primitivas de rastreamento força os programadores a criarem suas próprias soluções exclusivas. Isso leva à criação de ferramentas de rastreamento muito específicas e não reutilizáveis para depurar a computação na rede. Mais importante, rastros produzidos por soluções específicas para o PDP provavelmente não serão interoperáveis com estruturas de diagnóstico de rastreamento existentes, por exemplo, Dapper[Sigelman et al. 2010] do Google. Ortogonalmente, as estruturas de rastreamento existentes não fornecem primitivas para gerar ou capturar dados de rastreamento em planos de dados programáveis.

Um desafio de pesquisa atual visa preencher a lacuna entre técnicas tradicionais para telemetria de redes e *frameworks* de rastreamento distribuído. Isso requer abordar execuções que cruzem a fronteira da aplicação distribuída para o plano de dados programável, capturando dados de rastreamento de PDPs e apresentando-os ao plano de aplicação por meio de uma abstração flexível e bem definida. Um dos primeiros esforços nessa direção é o P4-Intel [Castanheira et al. 2019], que (i) aproveita a telemetria de rede para instrumentar PDPs no monitoramento de dados de rastreamento arbitrários definidos pelo usuário e (ii) coordena o armazenamento, coleta e formatação desses dados de rastreamento internamente, fornecendo apenas dados de contexto bem formados para qualquer ferramenta de depuração do plano de aplicação.

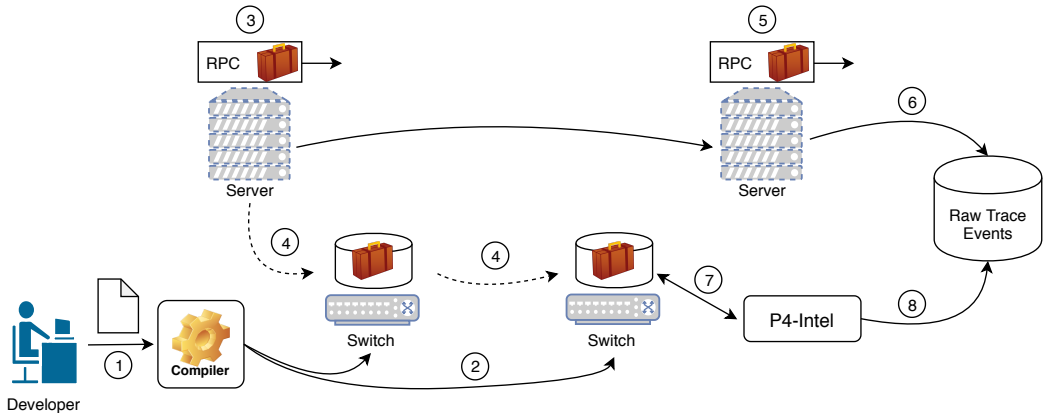

Figura 4.23: *Workflow* de Diagnóstico do P4-Intel. Malas vermelhas representam os contextos (*bagagem*), ou seja, os *logs* de rastreamento de RPCs. Os servidores armazenam o contexto dentro dos RPCs e propagam o contexto *in-band* dentro das mensagens RPC. Os *switches* armazenam os contextos localmente e os propagam *out-of-band*, em mensagens separadas, para o armazenamento externo.

A Figura 4.23 apresenta o *workflow* de alto nível do P4-Intel. Nele, de forma offline, desenvolvedores de aplicações (etapa 1) irão fazer anotações em programas de

in-network computing (escritos na linguagem P4) com o conjunto de variáveis definidas pelo usuário para exportar e implantar esses programas na rede (etapa 2). Essas variáveis que compõem o conjunto de dados que podem ser capturados e exportados usando uma abstração de *bagagem*. Em tempo de execução, o *framework* de rastreamento incluirá tags (ou cabeçalhos) em pacotes contendo chamadas RPC (etapa 3) e os propagará para outros servidores (etapa 5) via RPCs e, finalmente, os armazenará em um banco de dados externo (etapa 6) assim que o rastreamento de RPC for concluído. Além disso, à medida que esses rastreamentos de RPCs se propagam pela rede, o *framework* do plano de dados captura os dados apropriados e os armazena localmente no switch (etapa 4) ou, se necessário, os anexa aos pacotes. Periodicamente, o P4-Intel irá interagir com o plano de dados para exportar esses dados para uma entidade centralizada (passos 7-8) que combinará os dados capturados no plano de dados com os dados coletados nos RPCs.

Sistemas como o P4-Intel não apenas simplificam o rastreamento de programas PDP, mas também, dada a interoperabilidade com *frameworks* de sistemas distribuídos emergentes, também tem o potencial de simplificar o gerenciamento e facilitar o processo de depuração feito pelos programadores.

4.8. Considerações Finais

Os avanços em SDN expandiram a capacidade de programar a rede em direção ao plano de dados, o que possibilita a separação entre o controle da rede e as funções de encaminhamento. Tal separação está mudando radicalmente o cenário de rede, ajudando a enfrentar a ossificação de rede. Neste contexto, o operador de rede pode determinar a lógica para processamento de pacotes e assim habilitar arquiteturas dinâmicas, e gerenciáveis. No entanto, acentua-se a dificuldade e serem garantidas as propriedades de segurança e de corretude em toda rede, considerando uma combinação da configuração mantida pelo plano de controle e os programas do plano de dados.

O presente capítulo discute abordagens de segurança para redes de computadores na era dos planos de dados programáveis. Inicialmente, foi apresentada uma revisão dos principais fundamentos de programabilidade do plano de dados, especialmente no que tange à P4. Em seguida, é descrito como o conceito de programabilidade do plano de dados pode ser usado para enfrentar desafios de segurança em redes de computadores. Por fim, tópicos avançados e propostas já desenvolvidas neste contexto, são apresentados.

Apesar da discussão apresentada no capítulo, novos tópicos podem ser discutidos em trabalhos futuros. A relação entre a segurança através da programabilidade do plano de dados e a restrição de recursos de rede e/ou dispositivos pode ser explorada. A ampliação de funcionalidade de segurança em tal plano pode contribuir para assegurar propriedades de segurança em ambientes desafiadores. Finalmente, a relação de aspectos de segurança em redes 5G/6G e a programabilidade do plano de dados também pode ser abordada. A evolução das redes de comunicação móvel apresenta diversos desafios de segurança os quais poderiam ser enfrentados com dinamicidade no plano de dados.

Referências

[Anstee et al. 2017] Anstee, D., Bussiere, D., Sockrider, G., and Morales, C. (2017). Worldwide infrastructure security report. *Arbor Networks Inc., Westford, MA, USA*.

[Antonakakis et al. 2017] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., et al. (2017). Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*, pages 1093–1110.

[Avizienis et al. 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

[Beckett et al. 2017] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. (2017). A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168.

[Benson 2019] Benson, T. A. (2019). In-network compute: Considered armed and dangerous. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 216–224, New York, NY, USA. ACM.

[Bhuyan et al. 2015] Bhuyan, M. H., Bhattacharyya, D., and Kalita, J. K. (2015). An empirical evaluation of information metrics for low-rate and high-rate ddos attack detection. *Pattern Recognition Letters*, 51:1–7.

[Birnfeld et al. 2020] Birnfeld, K., da Silva, D. C., Cordeiro, W., and de França, B. B. N. (2020). P4 switch code data flow analysis: Towards stronger verification of forwarding plane software. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8.

[Bosshart et al. 2014a] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014a). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

[Bosshart et al. 2014b] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014b). P4: Programming protocol-independent packet processors. 44 (3): 87–95, july 2014.

[Bosshart et al. 2013] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. (2013). Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110.

[Boutaba and Aib 2007] Boutaba, R. and Aib, I. (2007). Policy-based management: A historical perspective. *Journal of Network and Systems Management*, 15(4):447–480.

[Brum 2022] Brum, H. B. (2022). Um método para coleta dinâmica e eficiente de estatísticas em redes programáveis.

[Carey 2017] Carey, S. (2017). Why a single failed router can ground a thousand flights. *The Wall Street Journal*.

[Castanheira et al. 2019] Castanheira, L., Schaeffer-Filho, A., and Benson, T. A. (2019). P4-intel: Bridging the gap between icf diagnosis and functionality. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ENCP '19, page 21–26, New York, NY, USA. Association for Computing Machinery.

[Charikar et al. 2002] Charikar, M., Chen, K., and Farach-Colton, M. (2002). Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer.

[Chow et al. 2014] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. (2014). The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, Broomfield, CO. USENIX Association.

[CiscoNetworks 2017] CiscoNetworks (2017). Cisco ios netflow. In *Accessed on June, 29 2017*.

[Craven et al. 2011] Craven, R., Lobo, J., Lupu, E., Russo, A., and Sloman, M. (2011). Policy refinement: Decomposition and operationalization for dynamic domains. In *2011 7th International Conference on Network and Service Management*, pages 1–9. IEEE.

[da Silva et al. 2015] da Silva, A. S., Smith, P., Mauthe, A., and Schaeffer-Filho, A. (2015). Resilience support in software-defined networking: A survey. *Computer Networks*, 92:189–207.

[Dang et al. 2016] Dang, H. T., Canini, M., Pedone, F., and Soulé, R. (2016). Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.*, 46(2):18–24.

[Dobrescu and Argyraki 2014] Dobrescu, M. and Argyraki, K. (2014). Software dataplane verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 101–114, Seattle, WA. USENIX Association.

[Feferman et al. 2018] Feferman, D. L., Mejia, J. S., Saraiva, N., and Rothenberg, C. E. (2018). Uma nova revolucao em redes: Programacao do plano de dados com p4. *Escola Regional de Informátiva do Piauí (ERIPI), Teresina, Brazil*.

[Fernandes and Rothenberg 2014] Fernandes, E. L. and Rothenberg, C. E. (2014). Openflow 1.3 software switch. *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC*, pages 1021–1028.

[Fosdick and Osterweil 2011] Fosdick, L. D. and Osterweil, L. J. (2011). Data flow analysis in software reliability. In *Engineering of Software*, pages 49–85. Springer.

[Freire et al. 2018] Freire, L., Neves, M., Leal, L., Levchenko, K., Schaeffer-Filho, A., and Barcellos, M. (2018). Uncovering bugs in p4 programs with assertion-based verification. In *SOSR*, page 4. ACM.

[Garcia et al. 2018] Garcia, L. F. U., Villaça, R. S., Ribeiro, M. R., Martins, R. F. T., Verdi, F. L., and Marcondes, C. (2018). Introdução à linguagem p4-teoria e prática. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)-Minicursos*.

[Ghasemi et al. 2017] Ghasemi, M., Benson, T., and Rexford, J. (2017). Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 61–74, New York, NY, USA. ACM.

[Gil and Poletto 2001] Gil, T. M. and Poletto, M. (2001). {MULTOPS}: A {Data-Structure} for bandwidth attack detection. In *10th USENIX Security Symposium (USENIX Security 01)*.

[Hamed and Al-Shaer 2006] Hamed, H. and Al-Shaer, E. (2006). Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3):134–141.

[Haq et al. 2015] Haq, O., Abaid, Z., Bhatti, N., Ahmed, Z., and Syed, A. (2015). Sdn-inspired, real-time botnet detection and flow-blocking at isp and enterprise-level. In *2015 IEEE International Conference on Communications (ICC)*, pages 5278–5283. IEEE.

[Hecht 1977] Hecht, M. S. (1977). *Flow Analysis of Computer Programs*. Elsevier Science Inc.

[Hoque et al. 2015] Hoque, N., Bhattacharyya, D. K., and Kalita, J. K. (2015). Botnet in ddos attacks: trends and challenges. *IEEE Communications Surveys & Tutorials*, 17(4):2242–2270.

[Horgan and London 1991] Horgan, J. R. and London, S. (1991). Data flow coverage and the c language. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 87–97.

[Jepsen et al. 2019] Jepsen, T., Alvarez, D., Foster, N., Kim, C., Lee, J., Moshref, M., and Soulé, R. (2019). Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 21â28, New York, NY, USA. Association for Computing Machinery.

[Jin et al. 2017] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. SOSP '17.

[Kang et al. 2016] Kang, M. S., Gligor, V. D., Sekar, V., et al. (2016). Spiffy: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. In *NDSS*, volume 1, pages 53–55.

[Kazemian 2017] Kazemian, P. (2017). Network path not found? *Forward Networks Blog*.

[Khurshid et al. 2013] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. (2013). {VeriFlow}: Verifying {Network-Wide} invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

[Kim et al. 2006] Kim, Y., Lau, W. C., Chuah, M. C., and Chao, H. J. (2006). Packetscore: a statistics-based packet filtering scheme against distributed denial-of-service attacks. *IEEE transactions on dependable and secure computing*, 3(2):141–155.

[Kreutz et al. 2013] Kreutz, D., Ramos, F. M., and Verissimo, P. (2013). Towards secure and dependable software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 55–60, New York, NY, USA. ACM.

[Kreutz et al. 2014] Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2014). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.

[Lakhina et al. 2005] Lakhina, A., Crovella, M., and Diot, C. (2005). Mining anomalies using traffic feature distributions. *ACM SIGCOMM computer communication review*, 35(4):217–228.

[Lapolli et al. 2019] Lapolli, C., Adilson Marques, J., and Gaspary, L. P. (2019). Offloading real-time ddos attack detection to programmable data planes. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 19–27.

[Liu et al. 2018] Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., and Foster, N. (2018). P4v: Practical verification for programmable data planes. In *ACM SIGCOMM 2018*, pages 490–503, New York, NY, USA. ACM.

[Lopes et al. 2016] Lopes, N., Bjørner, N., McKeown, N., Rybalchenko, A., Talayco, D., and Varghese, G. (2016). Automatically verifying reachability and well-formedness in p4 networks. *Technical Report, Tech. Rep*.

[Lopes et al. 2015] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., and Varghese, G. (2015). Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA. USENIX Association.

[Lupu and Sloman 1999] Lupu, E. C. and Sloman, M. (1999). Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869.

[Mace and Fonseca 2018] Mace, J. and Fonseca, R. (2018). Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 8:1–8:18, New York, NY, USA. ACM.

[McKeown et al. 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74.

[Mirkovic et al. 2002] Mirkovic, J., Prier, G., and Reiher, P. (2002). Attacking ddos at the source. In *10th IEEE International Conference on Network Protocols, 2002. Proceedings.*, pages 312–321. IEEE.

[Moshref et al. 2013] Moshref, M., Yu, M., and Govindan, R. (2013). Resource/accuracy tradeoffs in software-defined measurement. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 73–78.

[Moshref et al. 2016] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. (2016). Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143.

[Neves et al. 2018] Neves, M., Freire, L., Schaeffer-Filho, A., and Barcellos, M. (2018). Verification of p4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 73–85, New York, NY, USA. Association for Computing Machinery.

[Neves et al. 2021] Neves, M., Huffaker, B., Levchenko, K., and Barcellos, M. (2021). Dynamic property enforcement in programmable data planes. *IEEE/ACM Transactions on Networking*, 29(4):1540–1552.

[P4.org 2018] P4.org (2018). Switch. https://github.com/p4lang/switch.

[Panda et al. 2017] Panda, A., Lahav, O., Argyraki, K., Sagiv, M., and Shenker, S. (2017). Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 699–718, Boston, MA. USENIX Association.

[Peng et al. 2004] Peng, T., Leckie, C., and Ramamohanarao, K. (2004). Proactively detecting distributed denial of service attacks using source ip address monitoring. In *International conference on research in networking*, pages 771–782. Springer.

[Phaal 2009] Phaal, P. (2009). sflow: Sampling rates. In *June 2009*.

[Rapps and Weyuker 1985] Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375.

[Sanvito et al. 2018] Sanvito, D., Siracusano, G., and Bifulco, R. (2018). Can the network be the ai accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, page 20â25, New York, NY, USA. Association for Computing Machinery.

[Sapio et al. 2017] Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M., and Kalnis, P. (2017). In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA. ACM.

[Sflow 2017] Sflow (2017). sflow.org - making the network visible. In *Accessed on June, 29 2017*.

[Shannon 1948] Shannon, C. E. (1948). A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423.

[Sigelman et al. 2010] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc.

[Sivaraman et al. 2015] Sivaraman, A., Kim, C., Krishnamoorthy, R., Dixit, A., and Budiu, M. (2015). Dc.p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 2:1–2:8, New York, NY, USA. ACM.

[Son et al. 2013] Son, S., Shin, S., Yegneswaran, V., Porras, P., and Gu, G. (2013). Model checking invariant security properties in openflow. In *2013 IEEE International Conference on Communications (ICC)*, pages 1974–1979. IEEE.

[Stoenescu et al. 2016] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. (2016). Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM 2016*, pages 314–327. ACM.

[Travassos et al. 1999] Travassos, G., Shull, F., Fredericks, M., and Basili, V. R. (1999). Detecting defects in object-oriented designs: Using reading techniques to increase software quality. *SIGPLAN Not.*, 34(10):47–56.

[Udupi et al. 2007] Udupi, Y. B., Sahai, A., and Singhal, S. (2007). A classification-based approach to policy refinement. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 785–788. IEEE.

[Vergilio et al. 1997] Vergilio, S. R., Maldonado, J. C., and Jino, M. (1997). Constraint based selection of test sets to satisfy structural software testing criteria. In *Proceedings 17th International Conference of the Chilean Computer Science Society*, pages 256–263.

[Verma 2002] Verma, D. C. (2002). Simplifying network administration using policy-based management. *IEEE network*, 16(2):20–26.

[Westerinen et al. 2001] Westerinen, A., Schnizlein, J., Strassner, J., Scherling, M., Quinn, B., Herzog, S., Huynh, A., Carlson, M., Perry, J., and Waldbusser, S. (2001). Terminology for policy-based management. Technical report.

[Winder 2020] Winder, D. (2020). Much of the internet went down yesterday: Here's the reason why. *Forbes*.

[Xiong and Zilberman 2019] Xiong, Z. and Zilberman, N. (2019). Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 25â33, New York, NY, USA. Association for Computing Machinery.

[Yu et al. 2011] Yu, M., Greenberg, A., Maltz, D., Rexford, J., Yuan, L., Kandula, S., and Kim, C. (2011). Profiling network performance for multi-tier data center applications. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[Zargar et al. 2013] Zargar, S. T., Joshi, J., and Tipper, D. (2013). A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069.

