Combatendo vulnerabilidades em programas P4 com verificação baseada em asserções

Lucas Freire, Miguel Neves, Alberto Schaeffer-Filho, Marinho Barcellos[1]

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{lmfreire, mcneves, alberto, marinho}@inf.ufrgs.br

Abstract. *Current trends in SDN extend the network programmability to the data plane through the usage of programming languages like P4. In this context, the chances of introducing errors, and consequently software vulnerabilities in the network significantly increases, since packets can be processed in a general fashion. Existing data plane verification mechanisms are incapable of modeling P4 programs or present severe restrictions in the set of modeled properties. To circumvent these limitations and make programmable data planes more secure, we present a P4 program verification approach based on the usage of assertions and symbolic execution. First, P4 programs are annotated with assertions expressing general correctness and security properties. Then, the annotated programs are transformed in C code and all their possible paths are symbolically executed to verify the inserted assertions validity. Results show that it is possible to prove an expressive set of properties in the order of seconds using the proposed technique.*

Resumo. *Tendências recentes em SDN estendem a programabilidade da rede ao plano de dados através do uso de linguagens de programação como P4. Nesse contexto, a chance de se introduzir erros e, por consequência, vulnerabilidades de software na rede aumenta significativamente já que pacotes passam a ser processados de maneira genérica. Mecanismos de verificação de planos de dados existentes são incapazes de modelar programas P4 ou apresentam grandes restrições no conjunto de propriedades modeladas. Para contornar essas limitações e tornar planos de dados programáveis mais seguros, apresentamos um método de verificação de programas P4 baseado no uso de asserções e execução simbólica. Primeiro, programas P4 são anotados com asserções que expressam propriedades genéricas de corretude e segurança. Em seguida, os programas anotados são transformados em código C e todos seus possíveis caminhos são executados simbolicamente a fim de verificar a validade das asserções inseridas. Resultados mostram que é possível provar um conjunto expressivo de propriedades na ordem de segundos utilizando a técnica proposta.*

1. Introdução

A programabilidade do plano de dados permite que operadores implantem novos protocolos de comunicação e desenvolvam serviços de rede com agilidade. Através do uso de linguagens de programação como P4, é possível especificar em poucas instruções quais e

como cabeçalhos de pacotes serão manipulados pelos diferentes dispositivos de encaminhamento da infraestrutura. Apesar da flexibilidade proporcionada por esse paradigma, uma série de desafios precisam ser tratados antes da sua ampla adoção.

Um dos principais desafios enfrentados diz respeito à segurança. Se por um lado P4 introduz um novo eixo de programação (o plano de dados da rede), por outro também eleva as chances de ocorrerem bugs causados por implementações incorretas de protocolos. Tais bugs podem facilmente se transformar em vulnerabilidades caso eles possam ser explorados no sentido de violar políticas de segurança da rede. Uma forma de combater esse problema é testando se a rede satisfaz propriedades de interesse utilizando métodos de verificação. Diversos mecanismos foram propostos nesse sentido [Son et al. 2013, Dobrescu and Argyraki 2014, Lopes et al. 2015, Lopes et al. 2016]. No entanto, nenhum deles é capaz de verificar propriedades de segurança em programas P4.

Com o objetivo de habilitar a verificação de propriedades genéricas de corretude e segurança em programas P4 (p.ex. isolamento, integridade de cabeçalhos e ausência de laços), nós propomos um mecanismo baseado em asserções e execução simbólica. Primeiro, o programa P4 é anotado com asserções que expressam propriedades de interesse. O programa anotado é então traduzido para um modelo expresso em linguagem C e uma máquina de execução simbólica percorre todos os possíveis caminhos de execução do modelo gerado. Nosso mecanismo é capaz de provar que uma propriedade é satisfeita verificando se nenhum caminho de execução viola as asserções que a expressam.

Nós prototipamos o mecanismo proposto utilizando a ferramenta Klee [Cadar et al. 2008] e o compilador de referência da linguagem P4, disponibilizado pelo *P4 Language Consortium*[1]. Resultados mostram que o mecanismo proposto é capaz de verificar propriedades de segurança (p.ex. integridade de cabeçalhos e ausência de amplificação de tráfego) na ordem de segundos para programas com até 15 tabelas. Além disso, um número razoável de asserções (cerca de 10) pode ser analizado sem impactar significativamente o tempo de verificação. Em resumo, esse artigo apresenta as seguintes contribuições: i) uma linguagem para especificação de propriedades de corretude e segurança (na forma de asserções) envolvendo programas P4; ii) um mecanismo de verificação de propriedades baseado em asserções e execução simbólica; e iii) a avaliação experimental do mecanismo proposto. Esse é o primeiro trabalho a abordar segurança em planos de dados programáveis.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta a linguagem de programação P4 e os principais conceitos relacionados a planos de dados programáveis. Na sequência, a Seção 3 traz uma descrição detalhada da solução de verificação proposta. A Seção 4 descreve a avaliação experimental realizada, enquanto uma visão geral do estado-da-arte é apresentada na Seção 5. Por fim, a Seção 6 expõe as principais conclusões e trabalhos futuros.

2. Linguagem P4 e Planos de Dados Programáveis

A linguagem de programação P4[2], proposta por [Bosshart et al. 2014], tem o objetivo de possibilitar a programação do plano de dados de dispositivos de rede (isto é, switches e

[1] p4.org

[2] Consideramos neste trabalho a versão $P4_{16}$ por ser a mais recente

roteadores) de maneira simples e independente de arquitetura. Dessa forma, novos protocolos de comunicação podem ser rapidamente implantados na rede com apenas algumas linhas de código.

Um programa P4 inclui basicamente definições de cabeçalhos, metadados, *parsers*, ações, tabelas, blocos de controle e objetos externos. *Parsers* descrevem o mapeamento dos bits de um pacote de entrada para os respectivos *cabeçalhos* declarados no programa. Uma vez mapeados, tais cabeçalhos podem ser manipulados por *tabelas* e *ações*. A sequência exata de tabelas e ações aplicadas durante o processamento do pacote é definida de forma imperativa por *blocos de controle*. *Metadados* permitem armazenar temporariamente informações de estado do pacote. Por fim, *objetos externos* atuam como interfaces para estruturas de dados e funções específicas de cada dispositivo de encaminhamento. Por exemplo, contadores disponíveis em switches programáveis são manipulados por programas P4 através de objetos externos. Cabe ao fabricante dos dispositivos implementar essas interfaces.

O código de um programa P4 é normalmente organizado em bibliotecas. Uma biblioteca padrão contendo definições de tipos básicos da linguagem é fornecida juntamente com seu *framework* de programação. Exemplos de tipos básicos envolvem *packet_in* e *packet_out*, utilizados para representar o pacote (isto é, o *stream* de bits) de entrada e saída, respectivamente. Outras bibliotecas podem ser livremente definidas por fabricantes de dispositivos e programadores de rede.

Para que possa ser executado por um dispositivo de rede, um programa P4 é traduzido por um compilador para instruções específicas do dispositivo alvo. Isso permite que um programa P4 seja executado por dispositivos baseados em CPU, GPU, FPGA, processadores de rede ou ASICs programáveis, por exemplo. No processo, o compilador P4 converte o código fonte do programa em uma representação intermediária, modelada por um grafo acíclico direcionado (em inglês: directed acyclic graph, ou DAG). Cada nodo do grafo, representando um elemento do programa P4, é então transformado nas respectivas instruções de baixo nível de acordo com o dispositivo alvo. Várias otimizações de código costumam ser aplicadas durante esse processo, permitindo por exemplo que dispositivos de rede processem pacotes a taxas mais elevadas.

3. Verificação de programas P4 Baseada em Asserções

Para entender melhor o tipo de problema que estamos tratando, a Figura 1 mostra um exemplo de vulnerabilidade em um programa P4. Esse programa especifica um pipeline de processamento de pacotes contendo duas tabelas (*tcp_table* e *acl_table*), invocadas dentro do bloco de controle *ingress*. De maneira equivocada, ambas as tabelas serão aplicadas somente sobre tráfego TCP, fazendo com que todo pacote UDP seja encaminhado sem que qualquer tipo de filtragem possa ser aplicado.

Assumindo que a política de segurança da rede proíbe esse tipo de prática, o programa em questão poderia ser utilizado como ponto de partida para a execução de diversos ataques. Apesar da correção para esse exemplo ser simples (basta mover a tabela que implementa a lista de controle de acesso para fora da estrutura condicional), a capacidade de identificar automaticamente essa e outras vulnerabilidades em programas P4 representa um desafio de pesquisa relevante, para o qual apresentaremos uma proposta de solução no restante desse artigo.

```
1 control ingress() {
2  apply {
3    ...
4    if (headers.ip.nextHeader == TCP) {
5      tcp_table.apply();
6      // ACL será aplicada apenas sobre tráfego TCP
7      acl_table.apply();
8    }
9    ...
10  }
11}
```

Figura 1. Exemplo de vulnerabilidade em um programa P4

3.1. Visão geral

A ideia-chave da proposta consiste em verificar modelos dos programas originais anotados com asserções simples, de forma a reduzir a complexidade do processo de testes como um todo. Nesse sentido, a Figura 2 mostra o fluxo de controle do processo de verificação. Primeiro, o programa P4 a ser verificado é anotado com asserções que expressam propriedades genéricas de interesse. Essas propriedades podem refletir uma política de segurança da rede ou simplesmente representar o comportamento esperado do programa. Uma vez anotado, o programa dá origem a um modelo expresso em linguagem C, criado por meio de um processo de tradução. Opcionalmente, regras de encaminhamento também podem ser utilizadas como entrada pelo tradutor. O modelo gerado é verificado por uma máquina de execução simbólica, a qual testa todos os caminhos de execução do programa para encontrar potenciais falhas ou vulnerabilidades. Se nenhuma das asserções inseridas é violada durante o processo de execução simbólica, o programa P4 é considerado seguro com relação às propriedades analisadas. Caso contrário, a respectiva violação é reportada para que as correções necessárias possam ser aplicadas. Na sequência, descrevemos em detalhe cada uma das etapas do processo de verificação proposto.

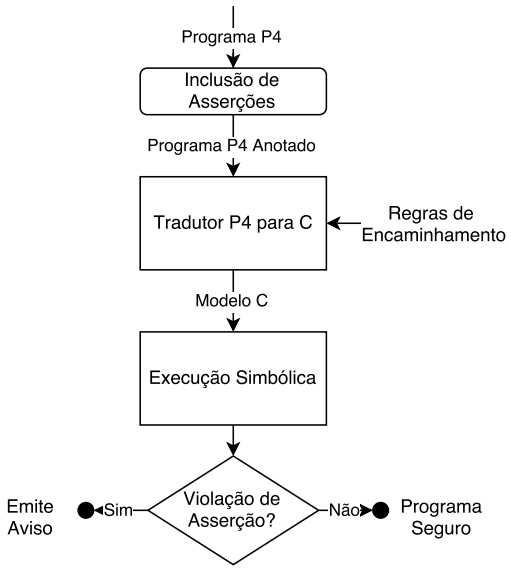


Figura 2. Fluxo de controle do processo de verificação.

3.2. Especificação de Asserções

Para incluir asserções em um programa P4, nós definimos uma linguagem de especificação que permite expressar propriedades de interesse a respeito de qualquer programa para plano de dados. Nossa linguagem faz uso do mecanismo de anotações de código disponível em P4 através da anotação *assert*, de tal modo que tanto o desenvolvedor do código quanto uma terceira pessoa podem expressar e/ou interpretar propriedades a respeito do programa de maneira intuitiva. A Figura 3 apresenta a sua gramática. De acordo com a figura, cada asserção a é composta de uma expressão e ou método $m \in \{forward, traverse\_path, constant, if, extract\_header, emit\_header\}$. Expressões e métodos, por sua vez, operam sobre um ou mais valores v, campos de cabeçalho f ou cabeçalhos h. O conjunto de métodos descrito foi escolhido de forma a facilitar a expressão das propriedades de segurança de maior interesse (p.ex., isolamento de rede, integridade de cabeçalhos, fluxo de informação, etc). Nosso objetivo não é permitir a especificação de qualquer propriedade. De qualquer forma, é possível facilmente estender a linguagem de asserções proposta no sentido de torná-la mais expressiva adicionando novos métodos.

Semanticamente, cada asserção representa um booleano que deve assumir valor lógico verdadeiro ou falso. Expressões têm a mesma semântica das suas equivalentes na linguagem P4, enquanto métodos possuem semântica própria. *forward* retorna verdadeiro quando o pacote sendo processado pelo programa é encaminhado para uma porta de saída válida. *traverse_path* indica se um dado trecho do programa foi executado, retornando verdadeiro sempre que essa asserção é de fato verificada. *constant(f)* é verdadeiro se o valor do campo *f* não é alterado durante a execução do programa. *if*(a_1, a_2, *[*a_3*])* possui semântica semelhante a declarações condicionais tradicionais: se a condição representada pela asserção a_1 for verdadeira, então a asserção a_2 será avaliada. Caso contrário, a asserção opcional a_3 terá sua validade verificada. *extract_header(h)* é avaliado como verdadeiro se bits do pacote de entrada são extraídos para o cabeçalho *h* durante o processo de *parsing* do pacote. Por fim, *emit_header(h)* retorna verdadeiro se o cabeçalho *h* for parte do pacote de saída ao final da execução do programa.

A Figura 4 mostra um exemplo de programa P4 anotado com asserções (em negrito). Por razões de espaço, somente as partes mais relevantes do programa são mostradas. Esse programa descreve um *pipeline* de processamento de pacotes com uma única tabela (*dmac*), instanciada dentro do bloco de controle *TopPipe*. Cada entrada dessa tabela pode invocar uma de duas ações (*Drop_action* ou *Set_dmac*). As asserções anotadas nesse programa buscam verificar: i) se pacotes marcados para descarte jamais são encaminhados (linha 7); e ii) se nenhum pacote com TTL igual a zero é encaminhado (linha 22).

3.3. Construção de Modelos C

Uma vez que o programa P4 é anotado, nosso mecanismo gera um modelo equivalente em linguagem C por meio de um processo de tradução. Essa seção descreve esse processo em detalhe, tomando como base as principais estruturas de um programa P4 (isto é, cabeçalhos, tabelas, ações, *parsers*, blocos de controle e objetos externos). O processo como um todo envolve duas etapas: i) transformação do programa P4 em um DAG; e ii) transformação do DAG gerado em código C a partir dos nodos de interesse. A Figura 5 exemplifica o processo de tradução das principais estruturas P4.

```
a ::= e | m
e ::= f
    | v
    | ! e
    | e && e
    | e || e
    | e == e
    | e != e
    | e >= e
    | e <= e
    | e + e
    | e - e
    | e * e
    | e / e

m ::= forward
    | traverse_path
    | constant(f)
    | if(a, a, [a])
    | extract_header(x)
    | emit_header(x)
```

Figura 3. Gramática da linguagem de asserções.

```
1  ...
2 control TopPipe(inout Parsed_packet headers,
3                 out OutControl outCtrl) {
4  ...
5  action Drop_action() {
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_action; Set_dmac; }
15   size = 1024;
16   default_action = Drop_action;
17 }
18 apply {
19   ...
20   dmac.apply();
21   ...
22   @assert("if(forward, headers.ip.ttl != 0)");
23 }
24}
```

Figura 4. Exemplo de programa P4 anotado com asserções.

Cabeçalhos. Cabeçalhos P4 são adequadamente modelados por *structs* em C, dado que possuem representações semelhantes. Cada campo do cabeçalho é mapeado para um membro da *struct*. Utiliza-se *bit fields* em C para manter a correspondência entre o tamanho do campo de cabeçalho e o tamanho do respectivo membro na *struct* gerada. Tal correspondência permite modelar comportamentos de *overflow* e *underflow* das variáveis P4. Cada tipo básico em P4 é mapeado para um tipo correspondente em C, considerando seu tamanho declarado. Note que apesar de P4 não impor um limite de tamanho para campos de cabeçalho, nenhum campo com mais de 64 bits pode ser modelado por essa abordagem devido a restrições nos tipos de dados suportados pela linguagem C. Planejamos investigar maneiras de representar campos com mais de 64 bits utilizando múltiplas variáveis ou *arrays* em C como trabalhos futuros.

Tabelas. Cada tabela de um programa P4 é modelada como uma função em C. Funções criadas a partir de tabelas são construídas de maneira diferente dependendo se as regras de encaminhamento são fornecidas para o tradutor ou não. Caso as regras sejam fornecidas, os campos de *match* da tabela P4 são testados contra os respectivos valores das regras utilizando a abordagem de casamento especificada (e.g., exata). Caso contrário, a decisão de qual ação executar é feita com base em um valor simbólico especialmente declarado para forçar a criação de múltiplos caminhos de execução para o programa pela máquina de execução simbólica (um para cada ação que uma regra na respectiva tabela poderia invocar).

Ações. Assim como tabelas, ações são modeladas como funções em C. Os parâmetros das ações devem ser traduzidos levando em consideração a estratégia de modelagem de tabelas. Quando as regras são desconhecidas, os valores dos parâmetros da ação também são desconhecidos. Nesse caso eles são tratados como variáveis simbóli-

	Programa P4	Modelo C

Cabeçalho	header ethernet_t {
	  bit<48> dstAddr;
	  bit<48> srcAddr;
	  bit<16> etherType;
	}
		typedef struct {
		 uint8_t isValid : 1;
		 uint64_t dstAddr : 48;
		 uint64_t srcAddr : 48;
		 uint32_t etherType : 16;
		} ethernet_t;

Tabela	table forward_table() {
	  actions = {
	  forward1;
	  NoAction;
	  }
	  key = {
	  hdr.ethernet.dstAddr: exact;
	  }
	  size = 32;
	  default_action = NoAction();
	}
		void forward_table() {
		  int symbol;
		  make_symbolic(symbol);
		  switch(symbol) {
		  case 0: forward(); break;
		  default: NoAction(); break;
		  }
		}

Ação	action forward(bit<9> port) {
	  standard_metadata.egress_spec = port;
	}
		void forward() {
		 uint32_t port;
		 make_symbolic(port);
		 standard_metadata.egress_spec = port;
		}

Bloco de controle	control ingress(inout headers hdr,
	  inout metadata meta) {
	  apply {
	  forward_table.apply();
	  }
	}
		// global variables
		headers hdr;
		metadata meta;
		
		void ingress() {
		  forward_table();
		}

Parser	parser TopParser(packet_in b,
	  out Parsed_packet hdr) {
	
	  state start {
	  transition parse_ethernet;
	  }
	
	 state parse_ethernet {
	  b.extract(hdr.ethernet);
	  transition select(hdr.ethernet.etherType) {
	  0x0800: parse_ipv4;
	  default: accept;
	  }
	 }
	
	}
		Parsed_packet hdr;
		
		void TopParser() {
		  make_symbolic(hdr);
		  start();
		}
		
		void start(){
		  parse_ethernet();
		}
		
		void parse_ethernet() {
		  hdr.ethernet.isValid = 1;
		  switch(hdr.ethernet.etherType){
		  case 0x0800: parse_ipv4(); break;
		  default: accept(); break;
		  }
		}

Asserção	action a1() {
	 @assert("traverse_path");
	}
		int path1 = 0;
		void a1(){
		  path1 = 1;
		}
		int main(){
		  ...
		  if(!path1) {
		  printf("Fail: traverse_path");
		  }
		  return 0;
		}

Figura 5. Exemplos de modelos C das principais estruturas P4.

cas. Se as regras são especificadas, então os parâmetros assumem os valores concretos especificados na regra equivalente.

Blocos de Controle. Como a declaração de um bloco de controle também inclui a declaração de suas ações e tabelas, os blocos acabam sendo traduzidos para múltiplas

funções C. Variáveis de escopo local do bloco de controle são declaradas como variáveis globais no modelo C para permitir que elas possam ser referenciadas por qualquer tabela e ação pertencentes ao bloco. O corpo do bloco tipicamente contém chamadas para tabelas e ações, que são modeladas como invocações de suas respectivas funções C.

Parser. Parsers são traduzidos para múltiplas funções em C: uma para o parser propriamente dito e outra para cada estado pertencente a ele. Como parâmetros e variáveis locais do parser podem ser acessados por qualquer estado em seu escopo, ambos são modelados como variáveis globais em C. Além disso, parâmetros de saída do parser (tipo *out*) ainda são declarados como variáveis simbólicas. Tais parâmetros representam os cabeçalhos do pacote, e portanto são equivalentes às entradas do modelo C. Sempre que o método externo *extract* (associado ao tipo básico *packet_in*) é encontrado em um estado do parser, o campo de validade do respectivo cabeçalho é configurado de acordo, deixando de ser uma variável simbólica. Em P4, ao final de cada estado do parser o fluxo de controle é transferido para o estado seguinte (comandos *transition* e *select)*. Nós modelamos esse comportamento em C através do encadeamento de chamadas de funcões, adotando elementos condicionais (isto é, *if* e *switch*) para as transições sempre que necessário.

Asserções. Cada tipo de asserção é modelada em C de maneira distinta. Operações booleanas e numéricas, bem como o método *if*, são diretamente traduzidas para construções equivalentes na linguagem C. Para modelar os métodos *extract_header*, *emit_header*, *traverse_path* e *forward*, é criada uma variável booleana global para cada ocorrência deles no programa P4. Tais variáveis assumem valor inicial falso e tem atribuído o valor verdadeiro em pontos diferentes do modelo de acordo com o respectivo método. Para o método *extract_header(h)* a atribuição ocorre logo após a invocação do método *extract* que recebe como parâmetro o cabeçalho *h* no programa P4. De forma similar, o método *emit_header(h)* faz com que a atribuição seja imediatamente após a invocação do método *emit* (associado ao tipo básico *packet_out*) cujo parâmetro é o cabeçalho *h*. Para *traverse_path*, a atribuição se dá logo antes da asserção que o contém. No caso do método *forward*, a atribuição ocorre ao final do *pipeline* de processamento de pacotes apenas se a porta de saída para qual o mesmo será encaminhado for válida. *constant(f)* é traduzido armazenando-se o valor do cabeçalho *f* em uma variável C no ponto da asserção, e testando se o valor dessa variável continua o mesmo ao final do programa.

Objetos externos. Esse tipo de estrutura é específico de cada dispositivo de encaminhamento, sendo somente sua interface utilizada por programas P4. Por essa razão, a modelagem do comportamento de cada objeto externo deve ser previamente conhecida. Na prática, isso significa integrar o modelo referente aos objetos externos de um dispositivo ao tradutor, por exemplo, por meio de bibliotecas.

3.4. Execução simbólica de modelos de programas P4

Após gerado pelo processo descrito na seção anterior, o modelo C de um programa P4 é verificado por uma máquina de execução simbólica. A execução simbólica de um programa requer que todos os seus possíveis fluxos de controle (isto é, seus caminhos de execução) sejam avaliados a partir de variáveis de entrada simbólicas. Nesse sentido, programas de planos de dados, e consequentemente seus respectivos modelos C, apresentam algumas particularidades específicas desse nicho de aplicação.

Essencialmente, programas P4 descrevem como um pacote de dados deve ser pro-

cessado ao ingressar em um dispositivo de encaminhamento, gerando ao final um pacote de saída ou então simplesmente descartando o pacote original. Nesse caso, os cabeçalhos do pacote que ingressa no dispositivo são considerados entradas do modelo, e portanto sempre assumem valores simbólicos. O número de caminhos de execução de um programa P4, por sua vez, é dado basicamente pela estrutura do seu *pipeline* de processamento de pacotes. Sempre que uma tabela pode ser acessada somente sob alguma condição (p.ex., dependendo do protocolo sendo utilizado), um novo caminho de execução é criado. O mesmo acontece sempre que mais de uma opção de ação pode ser invocada por uma mesma tabela.

4. Avaliação Experimental

Essa seção descreve a avaliação do mecanismo de verificação proposto. Nosso principal objetivo é responder às seguintes questões: (i) é possível encontrar vulnerabilidades em programas P4 utilizando o mecanismo proposto? (ii) quão expressivo é o mecanismo em termos do conjunto de propriedades de segurança que pode ser verificado? e (iii) quão eficiente é o mecanismo (isto é, em quanto tempo é possível verificar propriedades de interesse de acordo com as características de um programa)?

O restante da seção está organizado como segue: a Seção 4.1 apresenta o protótipo implementado bem como o ambiente de testes utilizado, enquanto as Seções 4.2 e 4.3 descrevem os estudos de caso realizados para avaliar tanto o potencial do mecanismo proposto como ferramenta de segurança quanto a sua capacidade de verificar um conjunto variado de propriedades. Por fim, a Seção 4.4 faz uma análise do seu desempenho.

4.1. Implementação e ambiente de testes

Nós prototipamos o mecanismo de verificação proposto utilizando a ferramenta Klee [Cadar et al. 2008] (versão 1.3.0) como máquina de execução simbólica. Para a construção de modelos C, primeiro convertemos um programa P4 para sua representação JSON utilizando o compilador de referência da linguagem, disponibilizado pelo *P4 Language Consortium*. Em seguida, traduzimos o modelo JSON (representado através de um DAG) para código C utilizando um tradutor implementado exclusivamente para esse propósito. O tradutor contém aproximadamente 750 linhas de código Python. Scripts Shell são utilizados para coordenar de forma automática a invocação de cada ferramenta no processo de verificação. O código fonte do tradutor, bem como os scripts e as cargas de trabalho utilizadas estão disponíveis online [3]. Todos os experimentos foram executados utilizando-se uma máquina virtual Linux (kernel versão 4.8.0) com um núcleo de 3 GHz e 16 GB de RAM.

4.2. Estudo de caso: DC.p4

Para mostrar o potencial do mecanismo de verificação proposto como ferramenta de defesa (isto é, prevenção) contra ataques a redes oriundos de vulnerabilidades em programas P4, selecionamos o programa DC.p4 como estudo de caso. DC.p4 foi proposto por [Sivaraman et al. 2015], e captura o comportamento de um switch de data center. Entre suas principais funcionalidades estão encaminhamento L2 e L3, VLAN, MAC learning, espelhamento de pacotes, tunelamento, listas de controle de acesso (ACLs) e balanceamento de carga (ECMP). Tal programa contém mais de 2500 linhas de código distribuídas

[3] https://github.com/ufrgs-networks-group/assert-p4

entre 37 tabelas. Essas tabelas, por sua vez, estão organizadas assumindo uma arquitetura com dois pipelines sequenciais de processamento de pacotes, um de entrada/ingresso e outro de saída/egresso, intercalados por um sistema de filas. Escolhemos esse programa pela sua complexidade e representatividade de cenários reais, e executamos verificações sobre o mesmo no intuito de encontrar falhas que pudessem ser exploradas em potenciais ataques.

Falhas em funcionalidades de segurança. O primeiro conjunto de experimentos realizado diz respeito à verificação de falhas nas funcionalidades de segurança implementadas pelo programa (p.ex. VLAN e ACL para prover isolamento de rede). Mais especificamente, verificamos se é possível desviar de alguma dessas funcionalidades de maneira a evitar que a respectiva política de segurança seja de fato aplicada sobre um pacote. Utilizamos asserções no formato *if(bypass == 0, traverse_path)* para expressar esse tipo de propriedade, onde *bypass == 0* indica que a funcionalidade de segurança está ativa (isto é, foi configurada pelo administrador), e *traverse_path* verifica se a mesma está sendo executada para todas as entradas possíveis.

Dada a complexidade do programa, o processo de verificação como um todo leva na ordem de dias para terminar (veja Figura 6(a)). Entretanto, nosso verificador foi capaz de encontrar violações das propriedades testadas em menos de 10s. Constatamos que apesar do programa oferecer múltiplas opções de listas de controle de acesso (p.ex. listas de níveis 2 e 3), pacotes IPv4 não passam por uma lista de nível 2, o que pode se transformar numa vulnerabilidade dependendo da configuração adotada pelo administrador da rede (p.ex. caso o administrador resolva configurar somente uma ACL de nível 2).

Falhas em replicações de tráfego. Algumas funcionalidades do programa DC.p4 (p.ex., espelhamento de pacotes) envolvem a replicação de tráfego entre portas do dispositivo ou para a sua CPU. Ambos os casos tipicamente fazem parte de alguma tarefa de monitoramento da rede. Se executados erroneamente, tais procedimentos podem servir de origem para ataques baseados em amplificação de tráfego [Krupp et al. 2016] ou replicação de pacotes [Lee et al. 2017] (p.ex., caso a réplica seja encaminhada ao mesmo destino do pacote original). Nesse sentido, verificamos se os procedimentos de replicação do programa DC.p4 são executados corretamente (isto é, se as réplicas são endereçadas às portas corretas ou se são devidamente modificadas antes do seu encaminhamento). Utilizamos asserções no formato ! *(outport == original_port && constant(outport))* dentro de ações que replicam o tráfego para expressar essas propriedades. O primeiro e o segundo membro que compõem essas asserções testam se a porta de saída do pacote replicado é a mesma do pacote original e se tal porta é alterada durante o processamento do pacote replicado, respectivamente.

Assim como no caso anterior, nosso mecanismo foi capaz de encontrar violações de propriedades rapidamente (menos de 13 minutos). De maneira geral, o programa é genérico o suficiente para permitir que o plano de controle configure a porta de saída tanto da réplica quanto do pacote original, ao mesmo tempo que nenhuma alteração de cabeçalho é feita sobre a réplica. Dessa forma, cabe ao plano de controle através da configuração das tabelas do dispositivo garantir que um mesmo pacote não seja encaminhado mais de uma vez para o mesmo destino, o que pode não ser uma tarefa simples.

4.3. Outros estudos de caso

Para avaliar a expressividade do mecanismo proposto em termos do conjunto de propriedades que podem ser verificadas, nós especificamos e testamos asserções para diferentes programas P4 utilizando a linguagem descrita na Seção 3.2. Nesse sentido, a Tabela 1 lista os diferentes programas testados, seus tamanhos (em linhas de código), as respectivas propriedades verificadas, o tempo de verificação em segundos e o consumo de memória observado (em MBs)[4]. Note que o tempo de verificação e o consumo de memória foram consideravelmente baixos para todos os casos.

Tabela 1. Estudos de caso

Programa	Tamanho	Propriedades	Tempo	Memória
VSS	226	Integridade do cabeçalho IPv4, pacotes com TTL igual a zero são descartados, pacotes marcados para descarte não são encaminhados	1s	< 2MB
MRI	278	Switch IDs adicionados em pacotes são autênticos, IDs adicionados não são removidos	2s	< 2MB

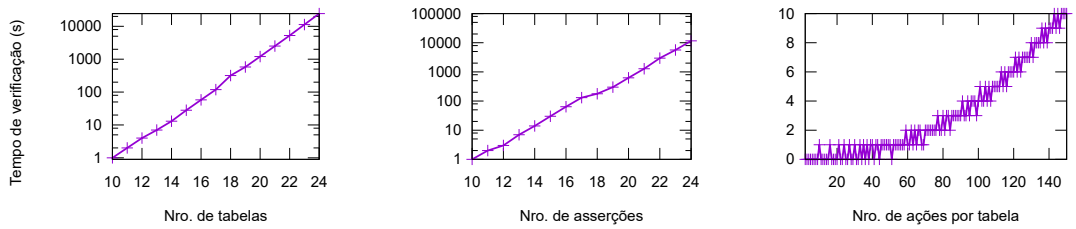
4.4. Análise de desempenho

Nessa seção, nós procuramos avaliar como o mecanismo de verificação proposto escala com relação a variações nas caracterísitcas de um programa P4. Para tanto, utilizamos o *benchmark* de planos de dados programáveis Whippersnapper [Dang et al. 2017] para gerar sinteticamente programas P4 com diferentes tamanhos de *pipeline* (isto é, quantidades de tabelas). Adicionalmente, testamos o impacto da variação: i) da quantidades de ações associadas com cada tabela; e ii) do número de asserções inseridas no código, para uma instância padrão de programa P4 gerada pelo *benchmark*, uma vez que esse não permite variar automaticamente o número de ações ou de anotações no código. Consideramos para esses casos um programa com somente duas tabelas (no intuito de minimizar o impacto desse fator no tempo de verificação), inserindo novas ações e asserções através de *scripts*.

Conforme esperado, o tempo de verificação cresce exponencialmente tanto para variações do número de tabelas quanto de asserções. Entretanto, é interessante notar que tal tempo é da ordem de algumas dezenas de segundos para programas com *pipelines* de até 15 tabelas (Figura 6(a)), o que inclui diversos programas P4 existentes atualmente [Dang et al. 2015, Sivaraman et al. 2017]. Ao considerarmos o número de asserções, podemos ainda perceber que verificações com mais de 20 propriedades podem levar algumas horas para serem concluídas (Figura 6(b)). Nesse caso, pode ser interessante verificar conjuntos diferentes de propriedades para um mesmo programa em execuções distintas, de forma que o tempo de verificação seja relativamente baixo para todas as execuções (verificações com até 10 propriedades não impactam significativamente o tempo de execução).

[4]VSS: https://github.com/p4lang/p4c/blob/master/testdata/p4_16_samples/vss-example.p4
MRI: https://github.com/p4lang/tutorials/blob/master/P4D2_2017/exercises/mri/solution/mri.p4

Por fim, podemos observar que o tempo de execução cresce de forma polinomial com a variação do número de ações (Figura 6(c)). Nesse caso, o grau do polinômio depende da quantidade de tabelas do programa P4 (p.ex., duas para o programa testado). Na prática, para um programa composto de p tabelas sequenciais, onde cada tabela pode invocar uma dentre q opções de ações diferentes, um total de q^p caminhos distintos devem ser executados simbolicamente. Pragmaticamente, programas P4 não costumam ter mais do que uma dezena de opções de ações para cada tabela [Dang et al. 2015, Sivaraman et al. 2017], o que torna o mecanismo proposto viável para a maioria das instâncias existentes.



(a) (b) (c)

Figura 6. Análise de desempenho do mecanismo proposto.

5. Trabalhos Relacionados

Verificação de redes. É possível encontrar na literatura diferentes abordagens de verificação que reforçam a segurança de planos de dados. Flover [Son et al. 2013] permite provar que o conjunto de regras instanciadas em switches OpenFlow satisfazem políticas de segurança da rede. Essa solução é inadequada para verificação de planos de dados programáveis devido a sua incapacidade de modelar comportamentos arbitrários de encaminamento. Para habilitar a verificação de redes com dispositivos programados em P4, recentemente foi proposto um mecanismo que cria automaticamente modelos em Datalog a partir de programas P4, de regras de encaminhamento e da topologia da rede [Lopes et al. 2016]. Os modelos criados são verificados com a ferramenta NOD (um resolvedor SAT) [Lopes et al. 2015] para provar propriedades relacionadas a alcançabilidade e formação correta de pacotes. Diferentemente desse trabalho, nossa abordagem permite verificar propriedades de segurança como integridade de cabeçalhos e ausência de amplificação de tráfego. Além disso, utilizamos uma linguagem de asserções que torna simples e intuitiva a especificação dessas propriedades.

Execução simbólica. A técnica de execução simbólica já foi usada anteriormente para verificar planos de dados. [Dobrescu and Argyraki 2014] prova que *pipelines* compostos por elementos Click satisfazem propriedades de *crash-freedom*, *bounded execution* e filtragem de pacotes. Os autores tentam lidar com o problema de explosão de estados executando simbolicamente os elementos Click de forma isolada. Symnet [Stoenescu et al. 2016], por sua vez, é um verificador de modelos de planos de dados construídos utilizando a linguagem SEFL, também proposta pelos autores. Tal linguagem contém instruções que simplificam a sua execução simbólica, permitindo a verificação eficiente de programas complexos. Apesar da sua escalabilidade, Symnet não pode verificar programas P4 em seu estado atual, uma vez que a linguagem SEFL não consegue

modelar todas as estruturas e manipulações de dados necessárias (p.ex., operações sobre bits).

Linguagem de asserções. Em [Beckett et al. 2014], os autores apresentam uma linguagem de asserções para verificar aplicações SDN. Com ela, é possível expressar propriedades que o plano de dados deve satisfazer em diferentes pontos de um programa de controle. As asserções são verificadas utilizando-se a ferramenta VeriFlow [Khurshid et al. 2012], que assim como Flover atua sobre regras de encaminhamento instanciadas em dispositivos OpenFlow. Enquanto a linguagem proposta por [Beckett et al. 2014] é utilizada em aplicações SDN, nossa abordagem é anotar diretamente um programa de plano de dados para provar propriedades de interesse.

6. Conclusão e Trabalhos Futuros

Apresentamos nesse trabalho uma linguagem de asserções que pode ser utilizada por programadores P4 para expressar propriedades de corretude e segurança de uma determinada implementação. Nossa solução é mais expressiva que outras abordagens de verificação de planos de dados, sendo o primeiro trabalho a permitir provar propriedades específicas de códigos fonte P4 e opcionalmente as regras de encaminhamento utilizadas por suas tabelas. Nosso mecanismo verifica as asserções utilizando execução simbólica sobre modelos C gerados automaticamente a partir do programa e asserções.

Avaliamos nossa abordagem provando propriedades de replicação de tráfego e contorno de funcionalidades de segurança do programa DC.p4, bem como diversas propriedades de programas encontrados na especificação e tutoriais de P4. A análise de desempenho do mecanismo proposto revelou que apesar de sua eficiência em verificar programas pequenos, o tempo de execução cresce exponencialmente com número de tabelas, ações e asserções. Por essa razão, em paralelo à abordagem aqui descrita, estamos trabalhando com os autores de SEFL na extensão da linguagem para habilitar a modelagem completa de programas P4. Este é um esforço de médio prazo, mas vislumbramos que através do Symnet será possível a verificação escalável de asserções em programas P4.

Referências

Beckett, R., Zou, X. K., Zhang, S., Malik, S., Rexford, J., and Walker, D. (2014). An assertion language for debugging sdn applications. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 91–96, New York, NY, USA. ACM.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Cadar, C., Dunbar, D., and Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA. USENIX Association.

Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. (2015). Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on*

Software Defined Networking Research, SOSR '15, pages 5:1–5:7, New York, NY, USA. ACM.

Dang, H. T., Wang, H., Jepsen, T., Brebner, G., Kim, C., Rexford, J., Soulé, R., and Weatherspoon, H. (2017). Whippersnapper: A p4 language benchmark suite. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 95–101, New York, NY, USA. ACM.

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.

Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. B. (2012). Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 49–54, New York, NY, USA. ACM.

Krupp, J., Backes, M., and Rossow, C. (2016). Identifying the scan and attack infrastructures behind amplification ddos attacks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1426–1437, New York, NY, USA. ACM.

Lee, T., Pappas, C., Perrig, A., Gligor, V., and Hu, Y.-C. (2017). The case for in-network replay suppression. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 862–873, New York, NY, USA. ACM.

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.

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.

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.

Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. (2017). Smoking out the heavy-hitter flows with hashpipe. *SOSR '17*.

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, R., Popovici, M., Negreanu, L., and Raiciu, C. (2016). Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 314–327, New York, NY, USA. ACM.

