Arquitetura de Virtualização Leve Multi-Tenant para Switches Programáveis

Ivan Peter Lamb, Pedro Arthur P. R. Duarte,
Marcelo C. Luizelli, José Rodrigo Azambuja, Weverton Cordeiro

Instituto de Informática – PPGC – Universidade Federal do Rio Grande do Sul
Porto Alegre – RS – Brazil

Abstract. *Virtualization is gaining traction in Programmable Data Planes (PDP), with several solutions in the literature for emulating or instantiating virtual programmable switches on the same host device. In this context, virtualization has numerous advantages, enabling multi-tenancy in programmable data center networks and greater device resource utilization. Nevertheless, enabling a complete multi-tenant solution requires security considerations not yet approached in previous investigations. In this paper, we present a PDP virtualization architecture based on program composition and access control for securely managing virtual switches from different tenants. Our experiments highlight the ability to transparently manage virtual switches hosted in the same physical device, in networking scenarios with multiple tenants.*

Resumo. *A virtualização está ganhando força em planos de dados programáveis (PDPs), com várias soluções na literatura para emular ou instanciar* switches *programáveis virtuais no mesmo dispositivo hospedeiro. Nesse contexto, a virtualização apresenta inúmeras vantagens, permitindo a multilocação (*multi-tenancy*) em redes de* data centers *programáveis e maior utilização dos recursos dos dispositivos. No entanto, habilitar uma solução* mult-tenant *completa requer considerações de segurança ainda não abordadas em investigações anteriores. Neste artigo, apresentamos uma arquitetura de virtualização para PDPs baseada na composição de programas e controle de acesso para gerenciar com segurança* switches *virtuais de diferentes* tenants*. Experimentos realizados destacam a capacidade de gerenciar com transparência* switches *virtuais hospedados no mesmo dispositivo físico, em cenários com vários* tenants*.*

1. Introdução

Os avanços proporcionados pelo paradigma de Redes Definidas por Software (SDN) e Planos de Dados Programáveis (PDPs) fornecem aos administradores de redes e pesquisadores uma grande flexibilidade sobre como um dispositivo do plano de dados (*switch*) interpreta e processa o tráfego a nível de pacotes [Michel et al. 2021]. Esta flexibilidade tornou possível a implementação de diversas funcionalidades e serviços diretamente no plano de dados, aproveitando-se da vantagem de processamento de pacotes na velocidade do enlace, que atualmente atinge a ordem de Tbps. As aplicações incluem *caching*, monitoração, armazenamento de chaves, consenso, etc. [Tokusashi et al. 2018, Dang 2019, Woodruff 2020]. Ao delegar parte do processamento de dados aos dispositivos de encaminhamento, também é possível reduzir a carga de trabalho dos servidores tradicionais e dispositivos na ponta da rede, que são os principais responsáveis por tais

tarefas no modelo tradicional. Mais importante ainda, o atraso da comunicação do plano de dados ao plano de controle (na ordem de milissegundos) é removido.

Recentemente, o interesse na virtualização de PDPs aumentou devido ao potencial de melhorar o aproveitamento dos *switches* programáveis físicos. Múltiplos *switches* lógicos (virtuais) podem ser implementados no mesmo dispositivo físico com esta tecnologia. Desta forma, o conceito de *hardware-slicing* também pode ser aplicado aos PDPs, similarmente a máquinas virtuais em datacenters tradicionais [Tanaembaum 2014]. A principal vantagem é a economia de recursos de hardware. Caso cada *switche* lógico fosse implementado em um dispositivo físico (ou seja, sem virtualização), o administrador da rede precisaria adquirir múltiplos dispositivos programáveis, cujo custo seria da ordem de dezenas de milhares de dólares.

No entanto, as soluções existentes carecem de discussões sobre os aspectos práticos envolvidos, além de não possuírem uma arquitetura fim-a-fim completa. Isto resulta na falta de adoção desta tecnologia pelo mercado, ou mesmo seu uso através de uma implementação de referência pela comunidade acadêmica. Sobretudo, faltam discussões sobre as garantias de isolamento dos *switches* virtuais e quanto a chamadas de gerência durante a operação.

A força dos trabalhos existentes está na base da virtualização, ou seja, nas técnicas para implementar os múltiplos *switches* virtuais. Quanto aos outros aspectos, eles mencionam ser necessário algum mecanismo externo para garantir o controle de acesso [Hancock and Van Der Merwe 2016, Zhang et al. 2017], sem detalhar qual; ou assumem apenas um *tenant* na rede e todos os *switches* virtuais pertencem a esse *tenant*, desprezando assim a necessidade de controle adicional [Zheng et al. 2020].

Além disto, a maioria dos trabalhos em virtualização foram avaliados em *switches* emulados por software, como o bmv2 [Hancock and Van Der Merwe 2016, Zhang et al. 2017, Zheng et al. 2020], que não é projetado para avaliação de desempenho. Para uma soluções adquirir adoção da indústria ela precisa ser avaliada em dispositivos utilizados na prática, ou seja, hardware capaz de prover processamento na velocidade do enlace (Tbps). Este é um passo fundamental para a adoção da tecnologia.

Visando retomar os avanços em virtualização de PDPs, neste artigo propõe-se uma arquitetura de virtualização baseada em composição de códigos de *switches* e controle de acesso. A arquitetura proposta permite (i) compor/compilar programas de *switches* distintos em um único programa, o qual emulará (virtualizará) esses *switches*, (ii) mapear os elementos do programa composto (por ex., tabelas de encaminhamento *match & action* e registradores) para um ou mais *switches* virtuais, e (iii) acessar *switches* virtuais de forma segura e transparente pelos *tenants* que possuam as credenciais e permissões necessárias. Por transparente, entende-se que cada *switch* virtual opera e pode ser gerenciado tal e qual um *switch* físico. O código-fonte da implementação de referência da arquitetura está disponível no GitHub para reprodutibilidade da nossa pesquisa [Authors 2024].

O restante do artigo está organizado como segue. A Seção 2 apresenta uma breve fundamentação teórica em redes definidas por software (SDN), planos de dados programáveis (PDP) e virtualização de PDPs. A Seção 3 descreve a arquitetura da solução proposta neste artigo e aborda os aspectos de controle de acesso e segurança da arquitetura. A Seção 4 descreve a avaliação experimental da implementação de referência da

arquitetura e, por fim, a Seção 5 conclui o artigo e fornece direções para trabalhos futuros.

2. Fundamentação Teórica

Nesta seção revisamos brevemente os principais conceitos que fundamentaram a pesquisa realizada, junto com algumas referências para maior aprofundamento dos tópicos.

2.1. Redes Definidas por Software

A introdução disruptiva do conceito de Redes Definidas por Software (SDN) permitiu uma maior configurabilidade dos dispositivos no núcleo da rede (inicialmente com protocolos fixos como OpenFlow [McKeown 2008] e, posteriormente, com Planos de Dados Programáveis). As inovações em redes de computadores desde então, sejam no âmbito de pesquisa ou indústria, possuem um objetivo intrínseco de prover aos operadores de redes uma maior flexibilidade dos objetos de hardware ou software de sua infraestrutura. Essencialmente, esta flexibilidade é provida pela divisão das funcionalidades da rede em:

- **Plano de Aplicação:** Formado pelas aplicações que interagem com a rede através de uma interface com o Plano de Controle;
- **Plano de Controle:** Traduz as requisições das aplicações feitas ao plano de dados e provê outras funcionalidades, entre elas autenticação e segurança. O plano de controle também é responsável por prover a visão abstrata do plano de dados para os serviços que executam no plano de aplicação;
- **Plano de Dados:** Onde os dispositivos de encaminhamento são provisionados, recebendo a lógica de processamento de tráfego do plano de controle. Os dispositivos podem ser provisionados via diferentes tecnologias como: ASICs, FPGAs, *switches* virtuais, entre outros. A padronização da interface com o plano de controle permite com que todas estas formas de implementação sejam compatíveis.

Entre as camadas apresentadas, a solução de virtualização proposta neste trabalho possui foco (i) no plano de dados, onde os dispositivos virtualizados são implementados e (ii) no plano de controle, que deve atuar em conjunto a cada *switch* virtualizado para prover uma abstração que permita o isolamento seguro dos dispositivos virtuais. A camada de aplicação, em última análise, deve "enxergar" os dispositivos no plano de dados de forma agnóstica à virtualização (ou seja, de forma transparente).

2.2. Planos de Dados Programáveis (PDPs)

Continuando a tendência de fornecer aos operadores e pesquisadores mais flexibilidade na configuração e gerência dos dispositivos de rede introduzida com o *OpenFlow*, consolidou-se a ideia de PDPs. Trata-se de um modelo onde a lógica de processamento dos pacotes do plano de dados é completamente customizável pelos operadores. Com PDPs, operadores podem desenvolver, implementar e testar novos protocolos e algoritmos de forma independente de especificações de terceiros. Portanto, os dispositivos migram de um paradigma *closed box*, em que os funcionamentos internos são determinados apenas pelos fabricantes e "impostos" aos operadores, para o paradigma *open box*, em que o funcionamento interno é aberto aos operadores pois eles próprios o definem.

Após isso, outros trabalhos foram realizados para a evolução e adesão deste novo conceito, em destaque a introdução da P4 (Programming Protocol-independent Packet

Processors) [Bosshart et al. 2014], que é uma Linguagem de Domínio Específico (DSL) para a definição do comportamento de um *switch* programável genérico. Essa linguagem possibilita agora a total programabilidade dos dispositivos, podendo-se implementar qualquer funcionalidade tradicional de dispositivos de rede, como firewalls, roteadores IP, balanceamento de carga, controle de congestionamento, entre outros, além de funcionalidades inovadoras de forma portável a diferentes hardwares. P4 teve ampla adesão do mercado, com diversos produtos comerciais disponiveis, entre eles Intel®Tofino ™, NetFPGA-SUME (Xilinx) e SmartNICs (Netronome e NVIDIA BlueField-2).

O presente artigo possui ênfase nas questões de segurança implementadas para gerência de *switches* virtuais. Estas funcionalidades, em grande parte, envolvem a interface de *runtime* entre o plano de controle e o plano de dados (que é a forma de acesso aos elementos dos *switches* durante a operação). Nossa solução é independente do *target* específico (seja hardware físico ou software de emulação de um dispositivo de encaminhamento), mas algumas modificações podem ser necessárias para adaptar a interface de *runtime* aos padrões definidos por cada fabricante.

Em nossa implementação de referência, utilizamos a interface P4Runtime [P4Org 2008], que provê métodos para acessar e modificar elementos do dispositivo programável durante sua operação como, por exemplo, inserir ou remover regras nas tabelas de encaminhamento ou ler contadores / registradores do dispositivo. Esta interface é open-source e implementada utilizando um servidor gRPC, que é um framework de Remote Procedure Calls (RPC). Este servidor normalmente executa dentro dos dispositivos físicos, instalados pelos fabricantes, e o plano de controle externo conecta-se então como um cliente para realizar as requisições.

2.3. Virtualização de PDPs

A ideia de virtualização dos dispositivos do plano de dados é inspirada em outros tipos de virtualização, como as populares Máquinas Virtuais [Tanaembaum 2014], que dividem um servidor *(host)* em múltiplas instâncias isoladas *(guests)*. A ideia fundamental é permitir que múltiplos programas (que definem a lógica de encaminhamento de um dispositivo do plano de dados) sejam implementados em um único dispositivo físico.

Técnicas de virtualização de dispositivos programáveis são recentes [Hancock and Van Der Merwe 2016, Zhang et al. 2017, Zheng et al. 2020], e não possuem soluções completas disponíveis a nível de mercado devido às limitações tais como desempenho, segurança e gerenciabilidade entre outras. Soluções que procuram um foco maior em gerenciamento, como o PvS [Bueno et al. 2022], não são completas em relação a prover uma arquitetura fim-a-fim de implementação e gerencia dos *switches* virtuais, dependendo de recursos externos para a implantação dos *switches*. A arquitetura de virtualização proposta no presente artigo permite a construção de uma solução completa que ataca essas limitações.

2.3.1. Virtualização de PDPs Baseada em Emulação

Os primeiros trabalhos em virtualização focaram em emular o comportamento de *switches* P4 utilizando um programa específico projetado para “acomodar” os programas dos *switches* virtuais. Esses programas contém ações e tabelas cuidadosamente formuladas

para que seja possível implementar as operações de um *switch* P4 qualquer via inserção de regras nestas tabelas durante a operação do dispositivo físico.

As duas principais investigações nesta linha são o Hyper4 [Hancock and Van Der Merwe 2016] e o HyperV [Zhang et al. 2017]. As principais limitações desses trabalhos são a falta de suporte a todas as operações existentes na linguagem P4, o tamanho limitado dos programas virtualizados devido ao número de estágios limitados dos dispositivos físicos disponíveis no mercado, e o grande impacto no desempenho dos *switches* virtuais.

2.3.2. Virtualização de PDPs Baseada em Composição

Outra abordagem para o problema da virtualização de PDPs foi introduzida com o P4Visor [Zheng et al. 2020]. A ideia foi utilizar composição de programas, ou seja, a virtualização ocorre a nível de compilação. Resumidamente, os códigos-fontes dos programas dos diferentes *switches* são "fundidos" em um único programa. Como são adicionadas poucas instruções e tabelas de *match & action* adicionais no processo de composição, o desempenho desta técnica é muito melhor do que a baseada em emulação.

O *tradeoff* desta abordagem é a perda de flexibilidade associada à possibilidade da implementação dos *switches* virtuais através de regras adicionadas a tabelas durante a operação [Hancock and Van Der Merwe 2016]. Para se adicionar e remover *switches* virtuais, é necessário interromper totalmente a operação do *switch* físico, executar novamente a ferramenta de composição, e realizar a re-implementação no *switch*.

Além disto, a composição de programas permite um melhor uso dos recursos através de virtualização leve (*lightweight virtualization*). Isto significa que os elementos dos programas são compartilhados quando possível. Tabelas de encaminhamento de programas distintos que compartilham uma mesma chave (por exemplo, uma tabela MAC de um *switch* da camada de enlace) podem ser unificadas no programa resultante. Em soluções *multi-tenant*, entretanto, é necessário que este compartilhamento seja realizado de forma transparente para prover isolamento. Ou seja, a visão lógica apresentada a cada *tenant* é que nada é compartilhado.

3. Virtualização Leve com Composição Genérica e Controle de Acesso

Como mencionado anteriormente, a falta de uma arquitetura completa limita o potencial da tecnologia de virtualização de PDPs. Nesta seção, apresentamos uma arquitetura completa de virtualização leve que utiliza os fundamentos dos trabalhos anteriores a fim de avançar a adoção da tecnologia.

A solução proposta considera que cada *switch* virtual (definido por um programa P4) pertence a um *tenant* distinto e suas funcionalidades podem ser completamente diferentes dos demais. Desta forma, os *tenants* podem fornecer seus programas independentemente, que poderão processar tráfego de protocolos diferentes.

A Figura 1 ilustra a arquitetura completa da nossa solução de virtualização. Múltiplos *tenants* podem implementar um *switch* virtual no sistema enviando programas P4 para o módulo de composição de código e, posteriormente, requisições ao módulo de *runtime*. O administrador do sistema possui uma interface para popular as políticas de

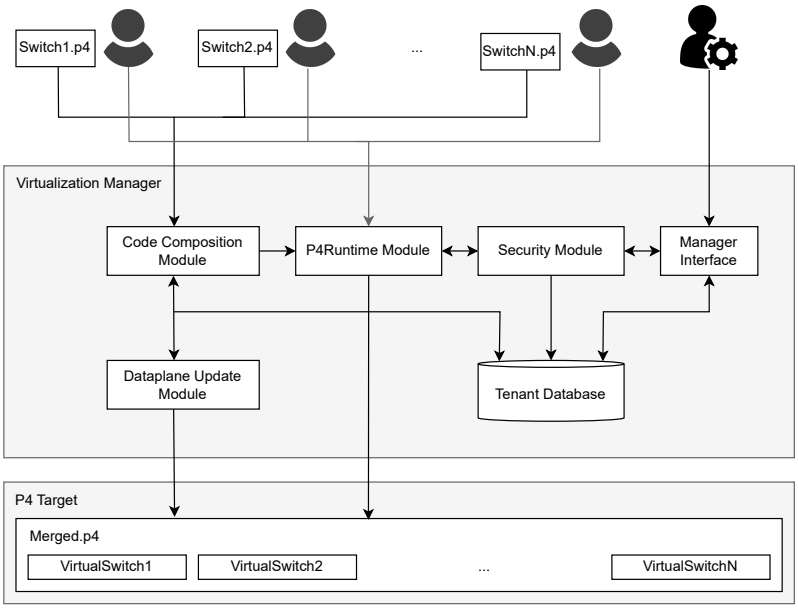


Figura 1. Arquitetura da solução de virtualização.

segurança e demais dados de cada *tenant* registrado no sistema. Os módulos ilustrados na figura possuem as seguintes funções:

(A) Code Composition Module: Responsável pela composição dos múltiplos programas P4 pertencentes a diferentes *tenants*. Este módulo gera um novo programa P4 que implementa a lógica de todos os programas fornecidos simultaneamente. O módulo transmite o programa composto ao módulo de *update* do plano de dados e também gera informações adicionais que serão utilizadas pelo módulo de *runtime* posteriormente. A Seção 3.1 descreve esta composição de forma mais detalhada.

(B) P4Runtime Module: A solução de virtualização deverá fornecer a impressão a cada *tenant* que seu programa P4 é o único programa executando no *target* para garantir o isolamento e segurança dos *switches* virtuais. O módulo implementa um servidor P4Runtime que se comporta como se fosse o servidor executando no "*target*" do *tenant*. Na realidade, o módulo intercepta as RPCs enviadas ao *target* e as "traduz" para serem compatíveis com o código composto que está efetivamente executando no *target*. Essa tradução é feita com as informações sobre os elementos compartilhados no código composto (providas pelo módulo de composição) e com restrições adicionais impostas pelo módulo de segurança.

(C) Security Module: Responsável por garantir que as requisições dos *tenants* estejam de acordo com as políticas de segurança definidas pelo administrador do sistema e por detectar outros possíveis ataques a *switches* virtuais.

(D) Manager Interface: Interface para que o administrador defina as políticas de segurança e popule a base de dados contendo as informações dos *tenants*.

(E) Dataplane Update Module: Neste módulo é realizada a implementação do programa P4 no *target*. Atualmente, este módulo simplesmente "troca" o programa que esta implementado quando requisitado, o que envolve procedimentos dependentes da arquitetura do *target*, como chamadas a compiladores e drivers específicos.

(F) Tenant Database: Banco de dados contendo informações dos *tenants* como, por exemplo, os recursos físicos disponíveis (número de portas, tamanho de memória, etc.), permissões de acesso às entidades do programa composto, entre outras.

O restante desta seção fornece mais detalhes sobre a estratégia de composição de programas do módulo de composição de código (3.1) e a abstração durante a operação dos dispositivos provida pelo módulo de *runtime* (3.2).

3.1. Composição de Programas P4

Como parte fundamental da contribuição deste artigo, propõe-se uma estratégia para composição de programas que privilegia aspectos de segurança. Além disto, de forma diferente dos trabalhos anteriores, a composição proposta é realizada diretamente nos programas P4, tornando-a completamente *target-independent*.

A composição necessita, no programa composto, de alguma informação que identifique a qual programa pertence cada pacote processado, direcionando o fluxo de controle de acordo com a lógica do programa específico. Essa informação é obtida pelo número da porta de entrada do pacote no *switch*. Cada porta utilizada no *target* possui um *tenant* associado através de um mapeamento definido pelo administrador.

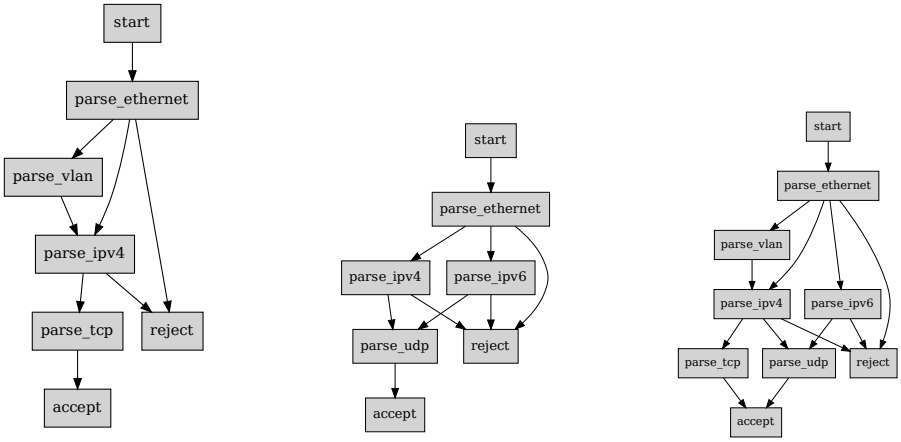
Além disso, a ideia de virtualização leve implica no compartilhamento de recursos do *target*. Os elementos da linguagem P4 que compartilhamos são (i) os estados dos *parsers* (blocos que extraem os cabeçalhos dos pacotes) e (ii) as tabelas *match & action* e as ações nos blocos de controle. Mais adiante descrevemos como a composição desse elementos é realizada considerando dois programas de entrada. Observe, no entanto, que a lógica pode ser facilmente estendida para suportar uma quantidade maior de programas.

3.1.1. Composição de Parsers

A composição de parsers ocorre alinhando-se as Máquinas de Estado Finitas (FSM) que representam os estados de parsing dos programas, unificando estados iguais. Esta unificação permite uma menor quantidade de estados do que simplesmente criarmos um novo parser com os estados de cada parser individual. Consideramos dois estados como sendo iguais se possuem os mesmos *statements*, à exceção do último, que indica a transição ao próximo estado. A transição é sempre modificada para se adequar à lógica de funcionamento dos parsers de entrada, utilizando a porta de entrada para selecionar o próximo estado. Desta forma, não precisamos de outro mecanismo de desambiguação de estados, como o implementado pelo P4Visor, que utiliza um sistema de *tags* adicional. O sistema de *tags* incorre na necessidade de estados artificiais introduzidos sempre que uma transição for ambígua e exige dispositivos adicionais na borda da rede para inserir/remover as *tags*. Em nossa solução, nenhum dispositivo físico adicional é necessário. A Figura 2 mostra um exemplo de como nosso sistema compõe dois parsers.

3.1.2. Composição de Blocos de Controle

Os elementos compartilhados no contexto de virtualização leve dos blocos de controle são as tabelas de *match & action* e as ações. Destes, as ações são os elementos mais simples



(a) (b) (c)

Figura 2. Exemplo da unificação de dois parsers (a) e (b) em (c).

de serem unificados, visto que são segmentos de código executados ao ocorrer um *match*, sendo que duas ações são unificadas caso sejam exatamente iguais. Em nosso modelo, consideramos que duas tabelas podem ser unificadas caso suas chaves de *match* forem iguais e sua ação padrão (tomada em casos de *table-miss*[1]) forem iguais e constantes. Os demais elementos descritos na especificação da linguagem P4 são a lista de ações e o número de entradas, que são trivialmente unificados através da união dos conjuntos de ações e da soma dos tamanhos das tabelas.

Não é possível selecionar todas as tabelas compatíveis para se realizar o *merge*, visto que o fluxo de controle resultante pode conter ciclos, o que não é permitido pela especificação da linguagem P4. Como uma primeira interação sobre este problema, neste trabalho escolhe-se um conjunto qualquer que satisfaça as restrições da linguagem.

Quando duas tabelas são selecionadas para serem unificadas, um novo elemento é adicionado à chave de *match* na tabela resultante, que serve para identificar o programa ao qual uma entrada pertence, visto que deve ser mantido o isolamento lógico das entradas de cada programa. A Figura 3 mostra um exemplo do resultado da composição das tabelas dos blocos de controle de dois programas distintos[2]. Na figura, a tabela **ecmp_group** do programa "a" é unificada com a tabela **ipv4_lpm** do programa "b" formando a tabela **ipv4_lpm** no programa composto "c". O nome das tabelas no programa composto é apenas para uso interno, sendo indiferente aos *tenants* que irão utilizar os nomes originais para referenciar-se a elas nas requisições de *runtime*.

3.2. Abstração de Runtime do Plano de Controle

O principal objetivo do módulo de *runtime* da nossa solução de virtualização é fornecer a cada *tenant* a visão lógica de que seu programa é o único executando no *target*. Levando

[1] *Table-miss* é o evento que ocorre quando um pacote não possui uma regra de *match* correspondente na tabela de *match & action*.

[2] Os códigos são trechos dos programas dos tutoriais de P4 sobre "basic forwarding" e "load balancing" disponíveis em: `https://github.com/p4lang/tutorials/tree/master/exercises`

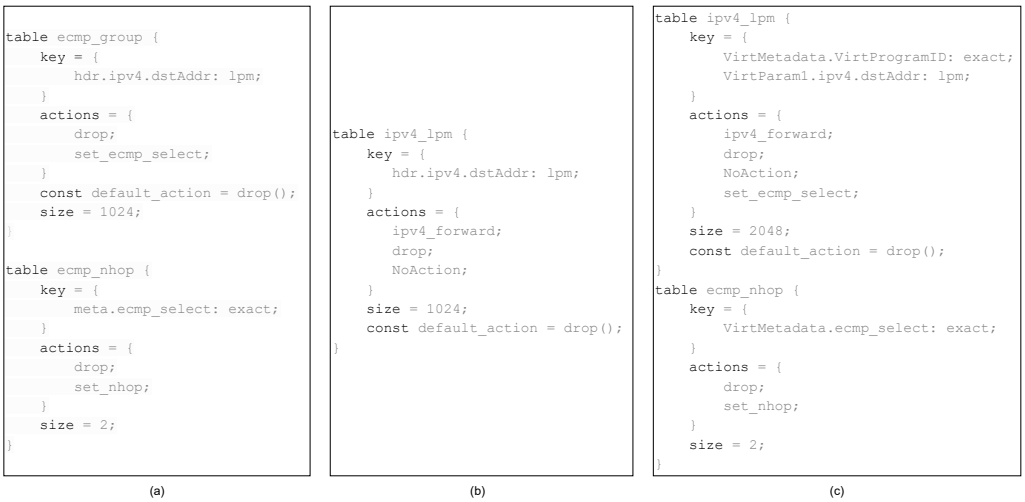


Figura 3. Exemplo da composição de tabelas de dois blocos (a) e (b) em (c).

em conta a especificação do P4Runtime, isso é feito fornecendo aos controladores de cada *tenant* um arquivo de informação (denomidado *p4info*) que lista o conjunto de tabelas de *match & action* e ações implementadas que podem ser manipuladas de forma similar ao arquivo que seria gerado caso seu programa fosse o único implementado.

O programa composto que de fato é implementado no *target* possui um conjunto de elementos descritos em seu arquivo de *runtime* (p4info), que podem ser provenientes de um único programa (recurso não compartilhado) ou o resultado da composição de elementos de múltiplos programas (recurso compartilhado). Durante a etapa de composição dos programas, é gerado um arquivo de mapeamento dos elementos de programas individuais para os elementos do programa compartilhado.

Ao receber uma RPC contendo uma requisição, o módulo P4Runtime gera uma nova requisição modificada para ser enviada ao *target*. De forma similar, a resposta é modificada para se adequar ao formato esperado pelo controlador de cada *tenant*.

Em nossa abordagem, onde os dispositivos virtuais podem ser logicamente independentes e isolados, é necessário prover segurança e controle de acesso a estes dispositivos. As principais necessidades são de (i) autenticar as requisições (ou seja, o *tenant*) e (ii) garantir o isolamento dos fluxos de processamento independentes, impedindo que algum *tenant* perturbe o funcionamento do *switch* virtual dos demais.

A necessidade de autenticação é provida através das funcionalidades de segurança provenientes do P4Runtime, que utiliza um servidor gRPC. Os servidores gRPC podem utilizar autenticação baseada em certificados TLS (Transport Layer Security). Aproveitando isso, a solução proposta utiliza esse esquema para autenticar os *tenants*. Certificados TLS são usados na comunicação de *runtime* para a autenticação dos clientes e segurança do canal de comunicação com criptografia. A comunicação entre o *Virtualization Manager* e o servidor P4Runtime executando no *target* também deve ser protegida com um mecanismo similar caso o canal seja inseguro.

3.3. Caso de Uso: Emulação de Topologias

No trabalho relacionado, P4Visor [Zheng et al. 2020], os autores enfatizam o teste diferencial de programas (versões parecidas de um mesmo programa) como caso de uso. Neste trabalho, apresentamos outra possível aplicação, desta vez aproveitando a possibilidade de múltiplos *tenants* no sistema.

Durante a fase de prototipagem de soluções inovadoras em PDPs, é necessário realizar testes em ambientes controlados. Para isto, softwares como o Mininet [Lantz et al. 2010] permitem instanciar *switches* emulados em software em um servidor convencional. Além disto, é possível definir topologias compostas de múltiplos dispositivos para testes mais completos.

A principal limitação destes ambientes decorre do uso de *switches* baseados em software, que possuem desempenho inferior a *switches* em hardware em ordens de magnitude. Protocolos que dependam de desempenho, como janelas de congestão, requerem ambientes mais próximos ao hardware onde serão implementados em produção.

Um dos casos de uso possíveis para a arquitetura apresentada é a emulação de topologias em hardware real. Através de uma extensão do módulo de *runtime* da arquitetura, é possível atribuir múltiplos *switches* virtuais a um *tenant*. Desta forma, grupos de pesquisa que possuam à disposição recursos limitados em termos de dispositivos programáveis podem utilizar de virtualização para emular topologias. Além disto, é possível que diferentes membros de um grupo de pesquisa executem simultaneamente múltiplos experimentos, pois a arquitetura permite isolar as topologias entre os *tenants*. A seção de resultados experimentais (4) mostra um caso de teste contendo a emulação de duas topologias isoladas executando em um único hardware.

4. Resultados Experimentais

Para validar a arquitetura proposta, desenvolvemos uma implementação de referência da arquitetura contendo os módulos fundamentais: (i) Code Composition, (ii) P4Runtime e (iii) Security. Realizamos três experimentos para verificar o isolamento e segurança dos *switches* virtuais, os custos adicionais no tempo de gerenciamento (em termos de latência), e o uso de recursos de hardware da solução proposta. Os experimentos têm como objetivo responder as perguntas: (i) O sistema implementado proporciona isolamento nos elementos implicitamente compartilhados? (ii) Qual é a latência adicional incorrida pelo *Virtualization Manager*? (iii) Quais são os potenciais ganhos em termos de uso de recursos de hardware oriundos da virtualização leve?

Utilizamos um *switch* programável Intel Tofino com throughput agregado de 6,4 Tbps para os testes de desempenho (latência) e uso de recursos e o *switch* de software de referência bmv2 para o experimento de isolamento. O uso do *switch* software demonstra que a arquitetura proposta é *target-independent*, podendo ser usada com diferentes dispositivos hospedeiros. A máquina utilizada para os testes com o *switch* de software possui as seguintes especificações: Ubuntu 22.04 (64-bits) com processador Intel® Core™ i7-8700 3.2GHz e 16GB RAM. O código-fonte da implementação de referência e *scripts* para reprodutibilidade da nossa pesquisa estão disponíveis no GitHub [Authors 2024].

4.1. Teste de Isolamento

Neste teste, a fim de avaliar a funcionalidade de isolamento e segurança da solução implementada, simulamos um cenário de um programa composto de três *switches* virtuais de *tenants* diferentes. Utilizamos o mesmo programa ("L3 forwarding") para todos os *tenants* para demonstrar que a solução é capaz de identificar corretamente as permissões de cada requisição e traduzi-las de acordo.

A topologia do experimento envolve os 3 *switches* virtuais implementados em um único *switch* do plano de dados. Cada um dos *switches* virtuais possui dois *hosts* conectados trocando dados entre si com *iperf*. A principal funcionalidade dos *switches* é encaminhar os pacotes baseados no IP de destino. Inicialmente, as tabela de encaminhamento possuem regras para encaminhar pacotes para os *hosts* conectados.

Os controladores de cada *switch* virtual são aplicações desenvolvidas para inserirem exatamente o mesmo conjunto de regras na tabela de encaminhamento de seus *switches* virtuais, sendo a única diferença aparente ao módulo P4Runtime as credenciais utilizadas para identificar os *tenants*. As aplicações inicialmente enviam requisições para remover todas as regras da tabela de encaminhamento, aguardam 10 segundos, e enviam requisições para inserir novamente as regras removidas. As requisições de RPC enviadas por cada aplicação são exatamente iguais, mas é esperado que requisições de um controlador específico afetem apenas os *hosts* conectados a portas de seu *switch* virtual.

O teste foi realizado iniciando cada controlador com 10 segundos de intervalo. A Figura 4 mostra os fluxos observados por *h2*, *h4* e *h6*, que são os receptores do fluxo *iperf* de cada *switch* virtual. Podemos observar que as regras inseridas por cada controlador não afetam o fluxo do *switch* virtual de outros. Isso ocorre devido às operações de composição, que adicionam o ID do programa virtual como chave em cada tabela.

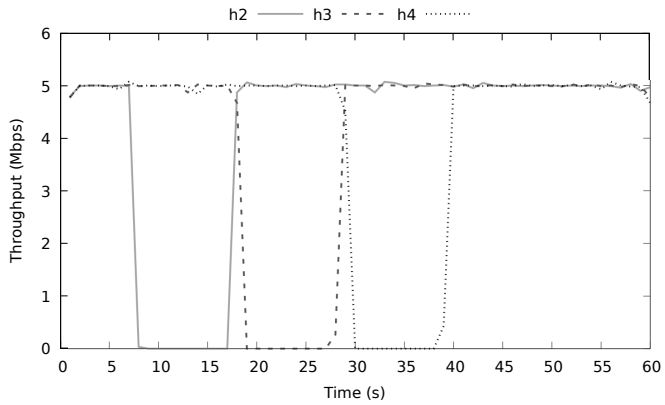


Figura 4. Fluxos observados pelos receptores no experimento de isolamento.

4.2. Teste de Latência

Para este experimento, utilizamos dois programas P4 que realizam funções diferentes. O primeiro programa processa tráfego baseado nos cabeçalhos IP como um roteador L3, porém pode responder a requisições de telemetria do controlador utilizando InBand-Telementry [Kim et al. 2016]. Os pacotes do controlador são populados com metadados especiais contendo informações como tamanho das filas, número de bits emitidos, etc.

O segundo programa também realiza o encaminhamento baseado em IP, porém as regras são inseridas reativamente pelo plano de controle quando ocorrem *table-misses*. Pacotes que não dão *match* na tabela de encaminhamento são enviados ao controlador juntamente com metadados informando a porta de entrada no *switch*. Este tipo de aplicação é útil para reduzir os tamanhos das tabelas, visto que somente regras referentes a fluxos ativos precisam ser mantidas. Um aviso de *timeout* é enviado do plano de dados para o controlador para avisar quando uma entrada de uma tabela não sofreu *match* em determinado intervalo. Uma resposta típica a estas mensagens é remover a entrada da tabela.

Os programas foram projetados para avaliar a capacidade da solução proposta de compor programas que impõem desafios, como (i) diferentes metadados de *packet-in* e *packet-out* (pacotes entre o plano de dados e o plano de controle), (ii) uma tabela compartilhada para reduzir o uso de hardware, (iii) lógicas de *parsing* e controle diferentes e (iv) mensagens em *runtime* de *timeout* que devem ser traduzidas para os controladores.

Durante o teste o controlador do *tenant* 1 periodicamente (a cada 3 segundos) envia pacotes de requição de telemetria ao *switch* virtual e registra o delay para receber um pacote contendo a resposta. O controlador do *tenant* 2 espera por pacotes que resultaram em *table-misses* ou por notificações de *timeout* e insere/remove regras de acordo. Ambos executam simultaneamente.

Tabela 1. Atraso medido no teste de latência.

	Com Virtualização (ms)	**Sem Virtualização (ms)**
C1 (Telemetria)	4.121	2.231
C2 (Forwarding)	19.09	11.06

A Tabela 1 mostra os atrasos para a aplicação de telemetria durante as requisições ao *switch* virtual 1 e o RTT (tempo de ida e volta) do primeiro pacote de um *ping* entre *hosts* conectados ao *switch* virtual 2. Para fins de comparação, a tabela também mostra os mesmos atrasos sem utilizar a solução de virtualização, executando os programas separadamente sem o *overhead* adicional. Os resultados são a média de 30 repetições dos experimentos. Apesar do delay das requisições quase dobrar com a solução de virtualização, as aplicações podem atingir praticamente o mesmo desempenho. A aplicação de telemetria irá observar um delay adicional de 2ms a cada 3s, muito provavelmente negligível. A aplicação de *forwarding* reativo irá precisar de cerca de 8ms de tempo adicional para estabelecer um fluxo, o que é desprezível para fluxos que duram minutos.

4.3. Teste de uso de recursos

Como mencionado anteriormente, virtualização leve permite com que recursos de hardware sejam compartilhados de forma transparente aos *tenants*. Isto permite um melhor aproveitamento destes recursos. Por exemplo, unificar tabelas diminui o uso total de hardware. Isto ocorre devido a otimizações no hardware, que permitem com que o uso de área de memória cresça de forma sub-linear conforme o número de entradas aumenta.

Para exemplificar o potencial ganho de recursos, realizamos a composição de 6 *switches* divididos em 2 topologias, conforme mencionado no caso de uso da arquitetura na seção 3.3. A Figura 5 mostra as topologias emuladas.

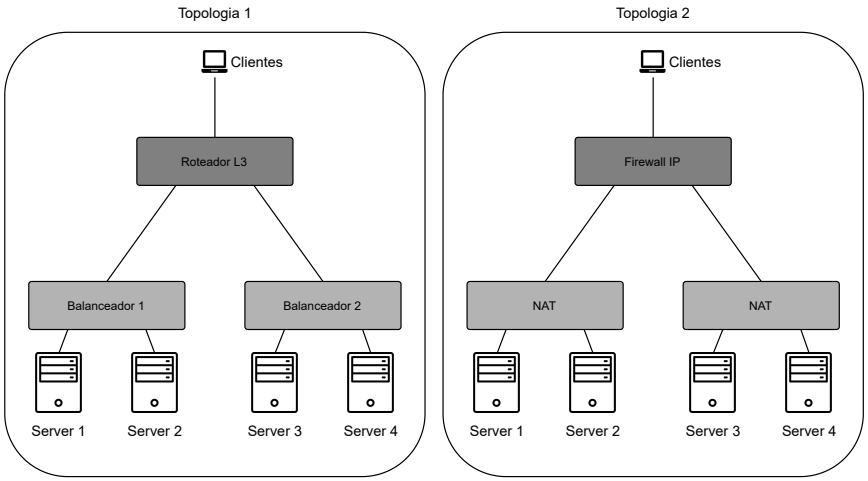


Figura 5. Topologias emuladas no teste de uso de recursos.

Tabela 2. Uso de recursos dos programas individuais e do programa composto.

Blocos de RAM	ipv4_t	firewall	src_t	dst_t	nhop	lb_table	Overhead	Total
Roteador L3	5							5
Balanceador					4x2	4x2		16
Firewall	4	4						8
NAT	4x2		4x2	4x2				24
Composto	14	4	8	6	4	5	5	46

A Tabela 2 mostra o uso de recursos, em termos de quantidade de blocos de RAM, de cada *switch* emulado quando seus programas são compilados individualmente. A tabela também mostra os mesmos valores para o programa composto. Os resultados mostram que é possível obter ganhos quando tabelas são unificadas. O número total de blocos utilizados (46) é 21% menor do que uma composição sem unificação de tabelas (58)[3].

5. Considerações Finais e Trabalhos Futuros

Neste trabalho, apresentamos uma solução completa de virtualização de PDPs com ênfase em segurança e isolamento dos dispositivos virtuais baseada em composição de códigos. Apresentamos também implementações das principais abstrações e interfaces que servem como base desta solução: o módulo compositor de código e a interface de *runtime*.

Em trabalhos futuros, pretendemos estender os mecanismos apresentados a fim de obtermos um produto final completo que poderá ser integrado com controladores amplamente utilizados, como o ONOS (Open Network Operating System). Ademais, são necessários estudos sobre estratégias que amenizem o impacto da reconfiguração do *host*.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001. Além disto, este estudo foi financiado em parte pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq).

[3]Equivalente à soma das tabelas individuais (53) e das tabelas de *overhead* da virtualização (5).

Referências

Authors (2024). Artigo SBRC secvirtpfour: Código-fonte do Projeto e Scripts. Url: `https://github.com/projectsecvirtpfour/secvirtpfour`.

Bosshart, P. et al. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Bueno, G., Saquetti, M., Rodrigues, P., Lamb, I., Gaspary, L., Luizelli, M. C., Zhani, M. F., Azambuja, J. R., and Cordeiro, W. (2022). Managing virtual programmable switches: Principles, requirements, and design directions. *IEEE Communications Magazine*, 60(2):53–59.

Dang, H. T. (2019). P4dns: In-network dns. *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

Hancock, D. and Van Der Merwe, J. (2016). Hyper4: Using p4 to virtualize the programmable data plane. In *CoNEXT'16*, pages 35–49. ACM.

Kim, C., Bhide, P., Doe, E., Holbrook, H., Ghanwani, A., Daly, D., Hira, M., and Davie, B. (2016). In-band network telemetry (int). *technical specification P*, 4:2015.

Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA. Association for Computing Machinery.

McKeown, Nick e Anderson, T. e. o. (2008). Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38:67–79.

Michel, O., Bifulco, R., Rétvári, G., and Schmid, S. (2021). The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Comput. Surv.*, 54(4).

P4Org (2008). P4runtime specification. ver. 1.3.0.

Tanaembaum, Andrew S. e Bos, H. (2014). *Modern Operating Systems*. Prentice Hall, 4th edition.

Tokusashi, Y., Matsutani, H., and Zilberman, N. (2018). Lake: The power of in-network computing. *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.

Woodruff, J. (2020). P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28.

Zhang, C., Bi, J., Zhou, Y., Dogar, A. B., and Wu, J. (2017). Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *Computer Communication and Networks (ICCCN), 2017 26th Int'l Conference on*, pages 1–9. IEEE.

Zheng, P., Benson, T. A., and Hu, C. (2020). Building and testing modular programs for programmable data planes. *IEEE Journal on Selected Areas in Communications*, 38(7):1432–1447.